conference

proceedings

The Third Conference on Object-Oriented Technologies and Systems (COOTS) Proceedings

Portland, Oregon
June 16–20, 1997

Sponsored by
The USENIX Association

The Advanced Computing Systems Association

USENIX Third Conference on Object-Oriented Technologies and Systems Proceedings (COOTS) *Portland, Oregon June 1997*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $20 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past COOTS Proceedings

COOTS II	June 1996	Toronto, Canada	$20/30
COOTS I	June 1995	Monterey, CA	$18/24

ISBN 1-880446-86-3

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

Third USENIX Conference

on

Object-Oriented Technologies and Systems

(COOTS)

June 16-19, 1997
Portland, Oregon

Conference Organizers

Technical Sessions Program Chair
Steve Vinoski, IONA Technologies, Inc.

Tutorial Program Chair
Douglas C. Schmidt, Washington University

Program Committee
Don Box, DevelopMentor
David Chappell, Chappell & Associates
Jim Coplien, Lucent Bell Labs
Murthy Devarakonda, IBM TJ Watson Research Center
Daniel Edelson, IA Corporation
Rachid Guerraoui, EPFL
Doug Lea, SUNY Oswego
Dmitry Lenkov, Hewlett-Packard
Mark Linton, Vitria
Stan Lippman, Walt Disney Feature Animation
Igor Metz, GLUE Software Engineering
Scott Meyers, Consultant and Author
Rajendra Raj, Morgan Stanley
Ron Resnick, IBM Israel
Vince Russo, Purdue University
Jonathan Shopiro, Novell
Joe Sventek, Hewlett-Packard Laboratories
Jim Waldo, JavaSoft

Tutorial Program Coordinator
Daniel V. Klein, USENIX

Conference Planner
Judith F. DesHarnais, USENIX

Table of Contents

The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)

June 16-19, 1997
Portland, Oregon

Wednesday, June 18, 1997

Thursday, June 19

Frameworks II
Session Chair: Daniel Edelson, IA Corporation

Security
Session Chair: Rajendra Raj, Morgan Stanley

Patterns
Session Chair: Doug Lea, SUNY Oswego

Panel Discussion: Reliable Distributed Object Systems
Moderator: *Jim Waldo, JavaSoft, Inc.*

Preface

Welcome to the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS). COOTS provides a forum for the presentation and discussion of the latest advanced research and development in the field of object technology.

This year's conference is special in that it marks what would have been the 10th anniversary of the USENIX C++ Conference, out of which COOTS evolved. The popularity of object technology today is largely due to the vast influence of C++, so it is therefore only fitting that Dr. Bjarne Stroustrup, the creator of C++, is our keynote speaker, and that quite a few members of the program committee have long been recognized as leaders in the C++ community. Even our conference site this year, the beautiful city of Portland, Oregon, was also once the site of a USENIX C++ Conference. However, this is not to say that C++ or any other programming language is the sole focus of this year's COOTS. Rather, the presentations you'll hear over the next two days will explore a broad range of topics, including distributed objects, frameworks, security, and patterns, with Java, C++, and even scripting languages being represented.

Another special feature of this year's COOTS program is that the final session on Thursday will be a panel on "Reliable Distributed Objects" that promises to raise discussions of many of the more difficult issues of practical distributed object systems. Distributed object computing has always been a very popular topic at COOTS, and this year is no exception.

I'd like to thank the program committee for their reviews of all the submitted papers and for their ideas for the conference program. I'd especially like to thank Doug Schmidt for coordinating our excellent COOTS tutorial program, and Joe Sventek for hosting the program committee meeting. I would also like to thank Rajendra Raj for organizing and chairing the Advanced Topics Workshop, and the Hewlett-Packard Company for their generous contribution in support of the workshop. Finally, I would like to acknowledge the sound advice and assistance that Doug Lea and Doug Schmidt gave me in my role as program chair.

I would also like to thank Ellie Young, Judy DesHarnais, Zanna Knight, Eileen Cohen, Toni Veglia, and Dan Klein at USENIX. They do all the hard work behind the scenes for the conference, such as planning and promotion. Please be sure to thank them for their efforts if you see them during the conference.

I sincerely hope that you thoroughly enjoy this year's COOTS and find it to be a valuable learning experience.

Steve Vinoski
IONA Technologies, Plc.

Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code

Gilles Muller, Bárbara Moura, Fabrice Bellard, Charles Consel
IRISA / INRIA-University of Rennes
Campus de Beaulieu
F-35042 Rennes Cedex - France
Fax: +33 2 99 84 71 71
Harissa@irisa.fr
http://www.irisa.fr/compose/harissa/harissa.html

Abstract

The Java language provides a promising solution to the design of safe programs, with an application spectrum ranging from Web services to operating system components. The well-known tradeoff of Java's portability is the inefficiency of its basic execution model, which relies on the interpretation of an object-based virtual machine. Many solutions have been proposed to overcome this problem, such as just-in-time (JIT) and off-line bytecode compilers. However, most compilers trade efficiency for either portability or the ability to dynamically load bytecode.

In this paper, we present an approach which reconciles portability and efficiency, and preserves the ability to dynamically load bytecode. We have designed and implemented an efficient environment for the execution of Java programs, named Harissa[1]. Harissa permits the mixing of compiled and interpreted methods. Harissa's compiler translates Java bytecode to C, incorporating aggressive optimizations such as virtual-method call optimization based on the Class Hierarchy Analysis. To evaluate the performance of Harissa, we have conducted an extensive experimental study aimed at comparing the various existing alternatives to execute Java programs. The C code produced by Harissa's compiler is more efficient than all other alternative ways of executing Java programs (that were available to us): it is up to 140 times faster than the JDK interpreter, up to 13 times faster than the Softway Guava JIT, and 30% faster than the Toba bytecode to C compiler.

Keywords: Java, C, Bytecode, Off-line compilers, JIT compilers

1 Introduction

The Java language [1, 2] provides a promising solution to the design of safe programs, with an application spectrum ranging from Web services to operating system components [3]. The success of Java is partly due to the fact that its basic execution model relies on the interpretation of an object-based virtual machine which is highly portable. However, the well-known tradeoff of Java's portability is the inefficiency of interpretation. Several solutions have been proposed to overcome this

[1]This research was supported in part by the Britany Council.

problem, such as just-in-time (JIT) [4, 5, 6, 7] and off-line [8, 9] bytecode compilers.

Just-in-time systems compile code to native form at runtime on demand. This approach avoids the overhead of compiling unused code, and eliminates the gap between compile time and execution time. Compiling during program execution, however, inhibits aggressive optimizations because compilation must only incur a small overhead. This is particularly important in the case of modern RISC processors for which complex analyses are required to achieve the best result. Moreover, the quality of the generated code critically relies on knowledge about the specific features of the target processor. Therefore, such compilers are not platform independent and requires a large amount of work to be ported.

Off-line compilers does not impose critical bounds on compilation time; optimizing analyses can be run as needed. They can also be platform independent, if they generate as output an intermediate language. However, in the context of Java, many applications dynamically load classes (i.e., bytecode) at runtime that limits applicability of pure off-line compilers.

In this paper, we present an approach that reconciles portability and efficiency, and preserves the ability to dynamically load bytecode. We have designed and implemented an efficient environment for the execution of Java programs, named Harissa[2]. Harissa provides a bytecode compiler and an interpreter integrated in the runtime library. Thus, a compiled program is still able to dynamically load classes and to interpret them. Harissa's compiler translates Java bytecode to C and furthermore incorporates aggressive optimizations.

To evaluate Harissa, we have conducted an extensive experimental study aimed at comparing the various existing alternatives to execute Java programs. The contributions of our work are as follows.

- The C code produced by Harissa's compiler is more efficient than all other alternative ways of executing Java programs (that were available to us): on the Caffeine Micro-benchmarks [10], it is 5 to 140 times faster than JDK 1.0.2 interpreter, 2 to 13 times faster than the Softway Guava JIT [6] and on average 20% faster than the Microsoft JIT compiler. On real application benchmarks, such as the *Javac* compiler, it is 5 times faster than the JDK interpreter and 30% faster than the Toba [9] bytecode to C compiler.

- The compiler statically evaluates the stack by abstractly interpreting the bytecode and replaces stack management with variables. This optimization suppresses one of the main sources of inefficiency in Java.

- The compilation process does virtual-method call optimization based on the class hierarchy analysis (CHA) [11, 12]. On the set of programs used in our benchmarks, this analysis permit the replacement of up to 40% of virtual methods calls by simple procedure calls.

- In contrast to existing off-line compilers, the runtime system of Harissa includes an interpreter that preserves the ability of an application to dynamically load bytecode.

- Finally, we discuss the benefits and limitations of off-line compilation *vs* JIT compilation. Based on our experimental study, we show that, for frequently used programs, it is always more advantageous to use off-line compilation rather than JIT compilation.

[2]Harissa was previously named Salsa.

The paper is organized as follows: Section 2 describes existing approaches for optimizing the execution of Java programs. Section 3 presents Harissa. Section 4 presents related work in class hierarchy analysis and existing bytecode compilers. Section 5 analyzes the performance of the code generated by Harissa's compiler on micro-benchmarks and real benchmarks, such as the *Javac* compiler and the *Javadoc* documentation generator. Section 6 concludes by describing future work and comparing JIT and off-line compilers.

2 How to Improve Java Execution

Several strategies have been presented to optimize execution of Java programs. They range from aggressive compilation schemes to specific hardware processors. Advantages and drawbacks of these schemes are the following:

- **Native Java compilers** - Compiling source code into native code is the most common way of compiling a language. But this approach is contrary to the Java philosophy since all the advantages of having an platform independent language disappear. For instance, the source code of Java programs is often not available. However, this strategy may be useful to obtain very efficient target binaries for very specific environments. This approach is implemented in the Vortex project [12].

- **Bytecode compilers** - Unlike native compiler, bytecode compilers take bytecode as input. One of the interesting characteristics of Java is that the bytecode contains nearly the same amount of information as the source itself. It has even been shown by Ford [13] and by Vliet [14] that it is possible to decompile the bytecode of a program and produce a Java source program similar to the original one. This is mostly due to the fact that the signature of the classes in the program must be kept in the bytecode to allow classes to be dynamically loaded at runtime. The only significant loss of information in the bytecode concerns structured loops, which are transformed into goto statements. Hence, a bytecode compiler can easily be as efficient as a native compiler. There are two types of bytecode compilers: those that generate native code and those that generate an intermediate language, such as C. The advantages of these two approaches are discussed below.

- **Just In Time compilers** - A just-in-time compiler differs from the a "classical" off-line compiler, in that the code is compiled only when needed at execution time. The difference in performance between those approaches is the time that can be spent during execution to perform optimizations. Vendors such as Borland [4], Symantec [5], Softway [6], and others have already released JIT compilers. The basic scheme is to compile a method when it is called for the first time, pausing execution while doing so. Refinement to this approach has been recently described by Plezbert and Cytron [7]. They mix interpretation and JIT compilation by taking advantage of multi-threading (on a multiprocessor)

- **Java Processors** - A Java processor is a dedicated processor that implements the Java Virtual machine and directly executes the Java bytecode. Such processor can be used as the main processor in a dedicated Java machine (workstations, embedded systems) or as a coprocessor in a workstation. Sun and

other manufacturers are already designing such chips. However, their competitiveness has not yet been proved [15]. Since such processors are not currently available, in this paper we only consider approaches that do not require specific hardware.

When is an Off-line Bytecode Compiler the Right Choice?

Although Java was originally designed for programming embedded applications, it has recently spread to many domains. Therefore, to choose the appropriate execution scheme many factors, such as the frequency of reuse of the same code or the heterogeneity level of the set of target machines, have to be considered. The most frequent situations are the following:

- **Small software components integrated in Web services** - These components can undergo frequent changes from one load to another by the same client. As a result, in this context, a JIT compiler is the most appropriate solution.

- **Platform-independent large software** - Such programs may or may not be related to Web services. Java technology is used because of its machine independence. The Java tools themselves are examples of such programs (e.g., compiler, disassembler, ...). These programs change infrequently and are often used by many users. Therefore, keeping a local, optimized version of the compiled code is advantageous. By comparison to a JIT, that always get the latest version of the software, this approaches requires the management of local optimized versions. This can be implemented by a revision control system that compiles and installs new software versions as they are released, in a automatic and transparent way.

- **Platform-dedicated software** - Examples are operating system components [3] and embedded applications. For these applications, the Java technology provides safety. These applications are characterized by very infrequent changes. Hence, it is advantageous to optimize the final code for the target system.

Finally, it should be noticed that even some statically configured tools, such as *Javadoc*, dynamically choose and load classes at execution time. For these applications, it is thus worthwhile to combine the binary code with an interpreter or a JIT compiler to allow dynamic (over)loading of new features.

Choosing C as a Target Output

As was already stated, there are two types of off-line bytecode compilers: native and non-native. Native compilers produce code that is directly executable, while non-native compilers produce code in an intermediate language.

Designing a native compiler has two advantages: (i) the generated binary code may be more efficient than that resulting from code written in an intermediate language and (ii) compilation is fast since it does not require successive tools. However, this choice has drawbacks: (i) it is not portable and (ii) generation of efficient code requires extensive knowledge of the features of the target processor.

Non-native compilers are more flexible and also achieve competitive performance. In particular, choosing C as an intermediate language permits the reuse of extensive compiler technology that has already been developed. In fact,

- There are very good C compilers.

- C compilers are available for all machines. The developer does not have to address subtle differences that exist between a processor and its successors.
- The development process is safer, quicker, and in some ways simpler since optimizations can be done on the generated C code.
- It is possible to reuse existing, aggressive optimizers such as Suif [16] or partial evaluators for C such as C-mix [17] or Tempo [18, 19].

These reasons led us to develop a non-native off-line compiler for Java bytecode that generates C programs.

3 Overview of Harissa

Harissa is a Java environment that includes a compiler from Java bytecode to C and a Java Virtual Machine integrated in a runtime library. While Harissa is aimed at applications that are statically configured, such as the *Javac* compiler, it is also designed to allow code to be dynamically loaded in an already compiled application. This novel feature is introduced by integrating a bytecode interpreter into the runtime library. Data structures between the Java compiled code and the interpreter are compatible and data allocated by the interpreter do not conflict with data allocated by the compiled code. Harissa is written in C and is designed with the primary goal of providing efficient and flexible execution of Java applications.

Because Harissa is written in C and its compiler generates C code, it is easily portable. In fact, current ports include SunOS, Solaris, Linux, and Dec Alpha. This allows us to compare the effects of optimizations on different architectures.

Because Harissa's compiler produces C programs, various compilers and optimizers can be used. As a result, contrary to JIT compilers, the generated C code does not have to be heavily optimized, since final optimizations are made by the C compiler. Harissa only concentrates on inefficiencies due to the architecture of the Java Virtual Machine: stack and method calls. To do so, several transformations are introduced. First, the stack is statically evaluated away. This analysis is described in section 3.3. Second, virtual method calls are transformed, when possible, into static (i.e., procedure) calls. For these virtual calls, type checks are also eliminated. This is described in section 3.4. Finally, Harissa implements several other optimizations for object-oriented languages such as method inlining, which are not presented.

The following sections describe the system in more detail.

3.1 Compiling a Java Program

Harissa's compiler takes as input a class C containing a `main` method and generates as output a `makefile`, a `_main.c` file, and a C source file for each class used in the program[3](see Figure 1). To determine the set of classes that depend on the initial class, an analysis is recursively performed on the bytecode to search for all the classes referenced by the main class. Because of the simplicity of this phase, it is omitted in the paper.

Compilation of a method's bytecode into C is organized as follows:

- Step 1 - The bytecode of the method is transformed into an intermediate bytecode representation (IBR). The purpose of this phase is to obtain a simpler and more regular representation. The IBR

[3]The system also supports separate compilation to reduce compilation time and the size of generated code. To do so, the compiler checks if a target class exits in the library before translating it. Since separate compilation conflicts with class hierarchy analysis and method call optimization, it has not been used in our benchmarks.

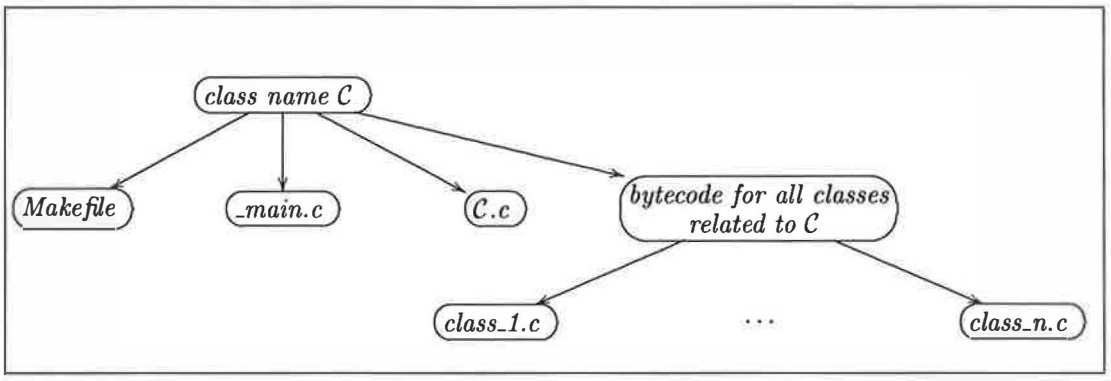


Figure 1: Set of generated C files given an initial class name C

simplifies the implementation of the subsequent passes by making explicit more detailed information than in the original Java bytecode.

- Step 2 - An analysis determines the value of the stack pointer before each instruction and the signature type for instructions that handle the stack. The result of this analysis allow the stack to be statically evaluated.

- Step 3 - A class hierarchy analysis is performed as described in [11]. This analysis permits the implementation of further optimizations on the intermediate representation. These optimization include: method inlining, transformation of virtual method calls into static (non-virtual) ones, and elimination of type checking. Method inlining and conversion of virtual method calls into non-virtual ones are iterated until no opportunities for further optimizations remain.

- Step 4 - This phase aims at eliminating bound checking. Checks are eliminated when it is possible to merge references to the same index as well as when the array bound and the index can be statically determined.[4]

- Step 5 - The final step generates the C code from the IBR. This phase is divided into three phases: (i) generation of goto labels and exception handling, (ii) declaration of local variables for the method, and (iii) translation of each intermediate bytecode instruction into C.

The following sections present our intermediate bytecode representation and the main algorithms that have an impact on performance. That is, the calculation of the stack pointer and the types of the stack instructions, and the transformation of virtual method calls into static procedure calls.

3.2 Intermediate Bytecode Representation

Our intermediate bytecode representation has a simpler and more regular syntax, and contains more detailed information than the original Java bytecode. The main difference is that, in the IBR, the types of the arguments of the instructions that handle the stack are

[4]This phase is not yet implemented in the current version of Harissa, although the code is conceived to include this optimization.

made explicit. This information simplifies subsequent passes.

The data structures defining the IBR are shown in Figure 2. A method contains information about its body and the exceptions that it can raise. Associated with each exception is the program counter of its handler. The `CodeInfo` structure has all the information about each instruction. Fields `in_sig` and `out_sig` represent the instruction's input and output signature, respectively. This explicit representation of signature types eases the subsequent analyses. The analysis described in the next section infers the type of instructions whose type is not explicit in the Java bytecode.

3.3 Calculation of the Stack Pointer and Instruction Signature Types

This analysis statically evaluates the stack by calculating the value of the stack pointer and the types of all the bytecode instructions. Most Java bytecode instructions have their type already associated with them, except those that control the stack. Because of the constraints enforced by the Java bytecode verifier [20], at each program point a stack instruction can have only one type signature. For example, when the instruction DUP is used to duplicate an integer, it can not be used at the same program point to duplicate a double. Thus, we can straightforwardly infer the types of the stack operations.

The analysis of a method via `CalculateSPandTypes` abstractly interprets each instruction with respect to a `Stack` structure (see Figure 2), which contains the current stack pointer value and the type of its items. `AnalyseCode` and `AnalyseExc` interpret the method's body and the code fragments corresponding to the method's exception handlers, respectively. The stack is initially empty. Abstract interpretation of an instruction can modify the contents of the stack. If an instruction I branches to more than one program point, then each branch is interpreted with respect to the stack resulting from abstractly interpreting I. Note that for the specific case of the jump to subroutine instruction (JSR), used to implement exceptions, the stack is assumed to be empty before and after the execution of the instruction. The JSR and RET instructions are considered to have the same control flow as a test instruction and the RETURN instruction, respectively. This approximation is not correct in terms of control flow information but gives correct results for stack type information.

Interpretation of an instruction is as follows: if the type of the instruction is not explicit in the Java bytecode, then the analysis has to infer it. The input signature is inferred from the types on the stack (function `infer_in_sig`). The output signature is inferred by abstractly interpreting the instruction with respect to this input signature (function `infer_out_sig`). Once the signature is known, then the instruction is abstractly interpreted with respect to the stack and its signature, with functions `pop_sig` and `push_sig`. The former checks for type consistency between the input signature and the type of the items it pops off the stack and the latter pushes the instruction's output signature onto the stack.

3.4 Transforming Virtual Calls into Non-Virtual Calls

Object-oriented programming encourages both code factoring and differential programming. This results in smaller procedures and more procedure calls. Procedure calls in an object-oriented language are dynamically dispatched. There are many analyses targeted at optimizing dynamically dispatched message sends. The most common are: intra-procedural static class analysis [11],

```
struct MethodInfo {
 CodeInfo *code;
 ExceptionInfo *einfo;
}

struct CodeInfo {
  char *in_sig, *out_sig;
  char opcode;
  list *instr_branch;
}

structure Stack =
  int sp;
  char *stack_type;
}

struct ExceptionInfo {
   ExceptionInfo *next;
   int handler_pc;
}

AnalyseCode (CodeInfo *code, int pc,
             Stack stk)
{
  CodeInfo instr;

  instr = get_instr (code, pc);
  if (visited? instr)
     return;
  else
     stk = AnalyseInstr (instr, stk);
  for each instr_branch do
  {
     stk' = stk;
     if (instr->opcode == JSR)
       stk' = empty_stk;
     AnalyseCode (code, branch, stk');
  }
}
```

```
CalculateSPandTypes (MethodInfo *minfo) {

  AnalyseCode (minfo->code, 0, empty_stk);
  AnalyseExc (minfo, minfo->code);
}

AnalyseInstr (CodeInfo instr, Stack stk) {
  char  *in_sig, *out_sig;

  in_sig = instr->in_sig;
  out_sig = instr->out_sig;
  if (unknown_sig? instr)
   {
     in_sig = infer_in_sig (stk);
     out_sig = infer_out_sig (in_sig);
   }
  stk = pop_sig (in_sig, stk);
  stk = push_sig (out_sig, stk);
  return stk;
}

AnalyseExc (MethodInfo *minfo, CodeInfo *code) {
  ExceptionInfo *einfo;
  Stack stk;

  einfo = minfo->einfo;
  while (einfo != NULL)
    {
      stk = push_item (REF, empty_stk);
      AnalyseInstr (code, einfo->handler_pc, stk);
      einfo = einfo->next;
    }
  }
```

Figure 2: Inferring instruction's type

class hierarchy analysis (CHA) [11], and profile-guided class receiver prediction [21]. In Harissa, we have opted to integrate a class hierarchy analysis to address this problem.

A class hierarchy analysis is a static analysis that determines a program's complete class inheritance graph (CIG) and the set of methods defined in each class. With the CIG, a specific set of possible classes, given that the receiver is a subclass of the class C, can be statically inferred and messages sent to the method's receiver can be optimized. Further, if there are no overriding methods in subclasses, a message sent to the method's receiver can be replaced with a direct procedure call and possibly inlined. Inlining of a method can trigger other opportunities for converting dynamic method calls into static ones. Hence, these two transformations are iterated.

3.5 Generation of C Code

The generation of the C code for a method is done in three phases. First, the goto labels and exception handlers are generated. Then, the local variables of a method are declared and, finally, each IBR instruction is translated to C.

Generation of goto labels and declaration of local variables are simple and are not discussed here. The treatment of exceptions needs some explanation. To ensure portability, Harissa handles exceptions in a stack-based manner. In the Java bytecode, each exception has a region associated to it. As described in the bytecode verifier documentation [20], different exception regions are either disjoint or nested, but cannot overlap. When translating the intermediate bytecode to C, entering of an exception region pushes the corresponding exception handler onto the stack, and exit of an exception region pops the exception handler off the stack. If a jump or goto instruction leaves an exception region or a set of nested exception regions, the corresponding exception handlers are popped off the stack prior to the jump or goto instruction.

The actual generation of the C code from the intermediate bytecode representation is straightforward. Figure 3-a shows some Java source code for a method computing a power function, Figure 3-b shows the corresponding Java bytecode. Figure 3-c shows the translated C code. In the C code, the stack has been statically evaluated: variable names prefixed with "s" are variables that handle the stack, while variable names prefixed with "v" are user-defined variables. An assignment to an `s`-variable corresponds to pushing a value on the stack. A use of an `s`-variable corresponds to popping a value off the stack. The `s`-variables can be eliminated either by a C compiler or by a C optimizer such as Suif [16]. Figure 3-d shows the optimized code generated by Suif.

3.6 Method Call Implementations

The implementation of a class includes a vector of function pointers that store the addresses of procedure implementing methods. Initialisation of this vector is performed when intantiating the class either at compile-time, by the compiler, or at run-time when dynamically loading byte-code. After initialisation, a pointer may refer either to a C procedure (i.e., method) of the compiled class, to a C procedure of an inherited compiled class, to a C native function of the run-time library, or to a `stubc` procedure. A `stubc` procedure interfaces compiled code with the interpreter: it allocates a stack for the interpreter, pushes arguments, calls the intrepreter's entry-point, and pops the result. `Stubc` procedures are generated by the compiler for each method that might be dynamically overloaded.

Interface calls are implemented using of a two dimensional sparse vector of function pointers for each class. The first dimension

```
static int P(int a,int b)
{
  int i,r;
  r=1;
  for(i=0;i<b;i++) r=r*a;
  return r;
}
```

a: Java source code

```
Method int P(int,int)
   0 iconst_1
   1 istore_3
   2 iconst_0
   3 istore_2
   4 goto 14
   7 iload_3
   8 iload_0
   9 imul
  10 istore_3
  11 iinc 2 1
  14 iload_2
  15 iload_1
  16 if_icmplt 7
  19 iload_3
  20 ireturn
```

b: Java ByteCode

```
TINT P(TINT vi0,TINT vi1)
{
  TINT si1,si0;
  TINT vi2,vi3;
  si0=1;
  vi3=si0;
  si0=0;
  vi2=si0;
  goto L14;
  L7:
  si0=vi3;
  si1=vi0;
  si0*=si1;
  vi3=si0;
  vi2+=1;
  L14:
  si0=vi2;
  si1=vi1;
  if (si0<si1) goto L7;
  si0=vi3;
  return si0;
}
```

c: C generated code

```
extern int P(int vi0, int vi1)
  {
    int vi2;
    int vi3;

    vi3 = 1;
    vi2 = 0;
    goto L14;
  L7:
    vi3 = vi3 * vi0;
    vi2 = vi2 + 1;
  L14:
    if (vi2 < vi1)
        goto L7;
    return vi3;
  }
```

d: Suif optimized code

Figure 3: Compilation of the power method

equals to the total number of interfaces referenced by the program, each interface being assigned an index at compile time. When a class is instantiated, if the class implements a given interface, the corresponding second dimension of the vector is allocated and is initialized with C procedures.

3.7 Current Status and Limitations of Harissa

Harissa is provided in two versions, with and without garbage collection (GC). This allows us to estimate the influence of GC on its performance. The GC version is based on the Boehm-Demers-Weiser conservative garbage collector [22]. The non-GC version relies on malloc, which leads to an increase in swapping and I/O since objects are never deallocated.

At the current time, threads are not implemented. Nevertheless, the system is already conceived to include them and the generated C code contains the necessary calls to synchronization functions. Implementation of synchronization optimizes the single thread case. As long as no additional threads are created, synchronization calls point to a null procedure. Additional threads creation is detected by guards [23] that then plug-in the multi-thread synchronization function.

For efficiency, Harissa produces a target C that relies on some gcc extensions. This is not a major limitation since gcc is available on many platforms. We plan to eliminate this dependency, in order to be able to test vendor C compilers. Finally, there are some native libraries, such as the graphic library, that are not yet supported.

4 Related Work

Other Off-line Compilers

To our knowledge, there are two other compilers from bytecode to C: J2C and Toba[5]. Harissa is the only environment that integrates an interpreter. J2C performs no optimizations when generating C code (i.e., stack evaluation or method call optimization). It is still immature and fails for many applications. Toba does a stack analysis similar to the one included in Harissa and generates C code from which transient variables have been eliminated. However, Toba does not do any method call optimizations. Currently, Toba is slightly more mature than Harissa since it supports threads.

Previous Work in CHA

Compilers for other object-oriented languages have included a CHA to optimize dynamically dispatched calls. In [24], Vortex, an optimizing compiler for object-oriented languages is presented. Vortex differs from Harissa in the following ways. It is a language-independent compiler with front-ends for Java, Cecil, C++, and Modula-3. Vortex takes as input source code. This approach limits its domain of use in the case of Java since source code is often not available. The optimizations it performs range from standard ones, such as constant propagation, dead code elimination, and method inlining, to optimizations specific to object-oriented languages, such as intraprocedural static class analysis, class hierarchy analysis [11], and profile-guided class receiver prediction [21]. The Vortex compiler has been used to study the impact of each of these optimizations alone and in combination. In [11], it is shown that class hierarchy analysis and profile-guided class receiver prediction are complementary transformations:

[5] Harissa and Toba have been developed independently at the same time.

the combination of the two produces a compounding effect.

Fernandez presents an optimizing linker that does class hierarchy analysis of Modula-3 programs [25]. Optimizations and code generation are done at link-time. The problem with this approach is that further optimizations that can result from transforming virtual calls into static procedure calls cannot be done by the compiler. An optimizing source-to-source C++ compiler is presented in [26]. The number of virtual method calls are reduced by performing both type feedback [27] and class hierarchy analysis. Method inlining is done as well. The optimized program is compiled by a native host C++ compiler.

5 Benchmarks

This section analyzes the performance gain that can be expected from an aggressive bytecode compiler. We compare execution of Harissa compiled programs with several industrial JIT compilers, the J2C and Toba bytecode compilers, and the JDK 1.0.2 interpreter.

Performance of JIT compilers is by nature sensitive to the target architecture since they compile into native code. To get more representative results, we have run the benchmarks on two different platforms: a Dell 100Mhz Pentium PC and a Sun 85 Mhz Sparcstation 5 (SS5). On the Pentium, Harissa is compared with the JIT compilers embedded in Netscape 3.0 and Microsoft Internet Explorer 3.0. On the Sparc, Harissa is compared with the Guava JIT compiler from Softway [6].

Three different kinds of benchmarks are presented: micro-benchmarks, which are used to evaluate the efficiency of JIT and off-line compilers for pure computations (without I/O); large benchmarks, which are used to compare JIT and off-line compilers for real applications that include I/O; and finally, benchmarks to evaluate the effectiveness of the CHA for Java applications.

Summary of results

Figure 4 summarizes our results. The micro-benchmark tests are made using Caffeine 2.5 [10]. Each Caffeine micro-benchmark tests one feature of the Java machine. On these tests, Harissa generated code is on average 50 times faster than JDK, 5 times faster than Softway Guava JIT [6] and 50% faster than Microsoft JIT.

On real application benchmarks, results depend mainly on how much pure computation the program does. On applications dominated by I/O, such as JHLZip and JHLUnzip, there is not much difference between off-line and JIT compilers; JDK is only 1.5 slower that Harissa. On applications such as *Javac* and *Javadoc* which rely on a mixed set of computation and I/O, Harissa is 5 times faster than JDK, 3 times faster than Softway Guava JIT and 30% faster than the Toba [9] bytecode compiler. On pure computation programs, such as an Othello game [28], Harissa is 2.6 times faster than Guava, 1.7 faster than Toba and 44 times faster than JDK. Toba results are missing when it was not possible to run it successfully, for reasons described below.

Methodology

Harissa has been configured so that during compilation, only methods with a size smaller than 100 instructions are inlined. The C code generated by Harissa and J2C has been compiled using gcc with the "-O2" option. The gcc version used is 2.7.2 on the Sun and 2.7.0 on the PC/Linux. Toba-generated C code has been compiled using Sun's commercial C compiler with the "-xO4" option.

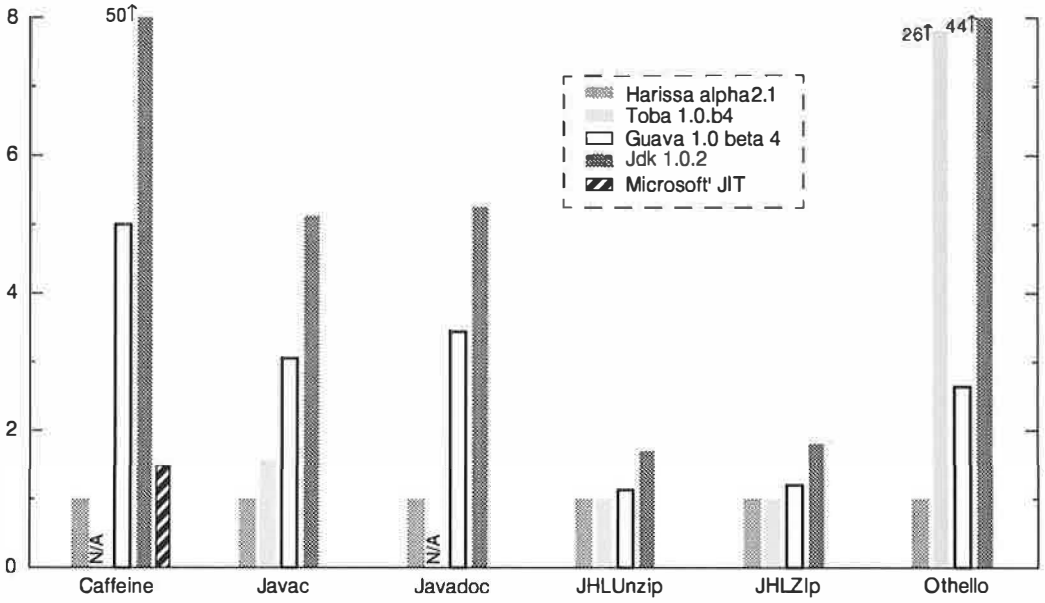


Figure 4: Execution time normalized to Harissa

5.1 Caffeine Micro-benchmarks

The Caffeine micro-benchmarks produce numbers, in CaffeineMarks (higher is faster), that allow one to compare heterogeneous architectures and Java implementations directly. Among them, we consider those that are related to the compilation scheme and that do not rely on graphic computations or the garbage collector:

- *Sieve* calculates prime numbers under 2048;
- *String2* tests string concatenation and search;
- *Logic* executes loops containing decision trees;
- *Loops* runs several types of integer loops;
- *Floating Point* (i.e., FP) simulates the calculations needed to rotate 50 three dimensional points by 90 degrees, 5 degrees at a time;
- *Method* tests how fast the VM performs method calls.

General comments about the results

The results of our evaluation are presented in Table 1 for the SS5 and in Table 2 for the PC. The two rightmost columns present Harissa's results with some further optimizations that are described below. In general, the PC is faster than the Sun. On the SS5, JDK and the interpreter embedded in Netscape achieve similar results, while the code generated by Harissa is 5 to 140 times faster than the JDK interpreter. On the PC, Microsoft's JIT compiler seems to be slightly faster than the Netscape's one, except for the tests *String2* and *FP*, which are twice as fast under Microsoft.

JIT compilers *vs* Harissa

The relevance of the micro-benchmarks when comparing JIT compilers and Harissa is to

	JDK	*Netscape*	*Guava*	*Harissa*		*Harissa+Suif*		*Harissa+Loop*	
	Cm *Interp.*	Cm *Interp.*	Cm *JIT*	Cm	ratio Guava	Cm	ratio Guava	Cm	ratio Guava
Loop	82	101	1657	11479	**6.92**	10170	**6.13**	-	-
Logic	95	116	968	9910	**10.23**	12935	**13.36**	-	-
String2	95	105	480	1390	**2.57**	1390	**2.57**	-	-
Method	82	75	102	460	**4.5**	460	**4.5**	-	-
Sieve	95	100	440	514	**1.16**	910	**2.06**	955	**2.17**
FP	83	92	544	970	**1.78**	1170	**2.15**	1295	**2.38**

Table 1: Comparison between JIT and Harissa on a 85Mhz SS5 (in *CaffeineMarks, Cm*)

measure the efficiency of the compilation scheme. Since the tests loop on the same code, JIT compilers do not loose time during execution waiting for the compilation of a method. Furthermore, with the exception of the *Method* test, no method calls are made. Harissa's inter-procedural optimizations such as CHA and method inlining thus have very little influence on the results. Therefore, these tests permit to evaluate precisely the quality of the code that is produced by JIT compilers.

Our measurements show that the code generated by Harissa'compiler is basically always faster than JIT compilers. Nevertheless, the results are architecture dependent. On the SS5, Harissa is 1.5 (for *Sieve*) to 13 (for *logic*) times faster than the JIT Guava. On the PC, results are more balanced and the difference in performance between Harissa and Microsoft is smaller than between Harissa and Guava, with a maximum of 2.5 times faster. For two tests, *Sieve* and *FP*, Harissa is actually twice as slow.

Improving the performance of the code generated by Harissa

To understand the reasons for the inefficiency of the code generated by Harissa for the tests *Sieve* and *FP*, we have analyzed the assembly code generated by gcc. For the *Sieve* test, it appears that the critical loop is about 20 instructions long. That does not leave much room for possible optimizations.

We have identified two reasons for inefficiency, which are in fact due to limitations of the gcc optimizer. As expected, transient stack variables are eliminated by gcc. But further optimizations resulting from variable and constant propagation are not triggered. For instance, in the *Sieve* test, stack variable elimination transforms a "`divide by i`" into a "`divide by 2`" that could then be efficiently transformed into a shift instruction. To evaluate the impact of this problem, we have used the Suif C optimizer [16] to systematically eliminate these variables using a combination of the "constant/variable propagation" and "dead code elimination" passes. The effect on the PC is dramatic for the *FP* and *Sieve* tests, nearly doubling the performance improvement. On the other tests there is little or no influence, which shows that this situation is not so frequent. On the Sparc, the influence of stack variable elimination is lower than on the PC. This is because the relative cost of processor instructions differs significantly between the Sparc and the Pentium.

A second source of inefficiency is the fact that loops are compiled into bytecode goto instructions. Therefore, gcc does not have all the necessary information to make the best choice regarding caching of temporary results in registers. To determine the consequences of this problem, we have reconstructed loops by hand for the *Sieve* and *FP* benchmarks.

	Microsoft Win 95	*Netscape Win 95*		*Harissa Linux*		*Harissa+Suif Linux*		*Harissa+loop Linux*	
	Cm	Cm	ratio Msoft	Cm	ratio Msoft	Cm	ratio Msoft	Cm	ratio Msoft
Loop	7087	7128	**1.005**	18000	**2.53**	18000	**2.53**	-	-
Logic	2032	1909	**0.93**	1930	**0.94**	2445	**1.20**	-	-
String2	1430	320	**0.22**	1833	**1.28**	1850	**1.29**	-	-
Method	2413	2028	**0.84**	4740	**1.96**	4790	**1.98**	-	-
Sieve	1370	1320	**0.96**	730	**0.53**	1420	**1.03**	1512	**1.1**
FP	2420	1306	**0.53**	1400	**0.57**	2350	**0.97**	2600	**1.07**

Table 2: Comparison between JIT and Harissa on a 100Mhz PC-Pentium (in *CaffeineMarks, Cm*)

On both the Sparc and the PC, there is a performance increase between 5% to 10%. Finally, it should be noted that after performing the optimizations, Harissa's compiled code is about 10% faster than Microsoft's JIT.

5.2 Real-Sized Benchmarks

These benchmarks are used to estimate the efficiency of Harissa in a real environment. To do so, we have evaluated the execution time of a set of programs that either do pure computations, substantial I/O, or a mixture of both. Pure computation programs are represented by an Othello game [28]. File handling applications (e.g., I/O) are represented by JHLZip and JHLUnzip, which insert and extract file from an archive without compression. Mixed computation-I/O programs are represented by two Sun's JDK tools, the *Javac* compiler and the *javadoc* documentation generator, and by *Kawa*, a scheme interpreter [29].

The benchmarks were made in a single-user environment to avoid external interferences. It was not possible to run benchmarks for JIT compilers embedded in the Web browsers for security protection reasons. Performance of tools such *Javac* and *javadoc* depends significantly on their input. To get representative results, we ran them on a set of large Java programs that are available on the net:

- Jas generates bytecode from a scheme based scripting language [30].
- Jax generates tokenizers from regular expressions [30].
- Jell generates a recursive descent parser from from a LL(1) grammar [30].
- Kawa is a scheme interpreter [29].

Comparisons are performed on real execution time, which includes waiting for the end of I/O, since this corresponds to what the user observes. For completeness, we have also detailed user and system CPU time spent during the execution to measure the efficiency of pure computations.

Detailed Javac results

Detailed timing of *Javac* execution are presented in Table 3. In comparison with JDK, Harissa achieves the highest speedup which is greater than 5. Toba is on average 3.3 times faster than JDK, J2C is about 2.5 times faster, and Guava is 1.5 times faster. These results clearly show the benefits of the various optimizations performed in Harissa.

We have also compared Harissa's GC version with the non-GC one. The GC version version is 20% faster than the non-GC one. This due to the fact that never reclaiming objects leads to an increase in swapping, I/O,

	JDK 1.0.2			*Guava 1.0 beta 4*				*J2C*			
	real mn	user cpu sec.	sys cpu sec.	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
Jax	0:44	37.2	2.5	0:29	16.5	4	**1.5**	0:16	7.7	2.6	**2.7**
Jell	0:47	41	2.5	0:30	18.4	4.4	**1.5**	0:17	8.9	3.5	**2.7**
Jas	1:45	69.5	9.0	0:59	37	9	**1.8**	0:41	15.6	7.6	**2.5**
Kawa	3:50	134	24	1:59	58	19	**1.9**	1:41	32	17	**2.3**

	Toba				*Harissa/no GC*				*Harissa/GC*			
	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
Jax	0:12	8	2	**3.6**	0:10	4.7	2.7	**4.4**	0:09	4	1.9	**4.9**
Jell	0:14	9.2	2.9	**3.3**	0:10	5.2	3	**4.7**	0:09	4.7	1.9	**5.2**
Jas	0:35	15	5	**3**	0:25	8.8	5.5	**4.2**	0:20	7.2	3.8	**5.2**
Kawa	1:10	28.7	10.5	**3.2**	0:58	17.6	11.5	**3.9**	0:44	14.5	8	**5.2**

Table 3: Compilation time of several Java programs

and in the amount of address space that has to be allocated by the system to the process.

Detailed Javadoc results

Javadoc is representative of tools that rely on the dynamic capabilities provided by Java to load bytecode during execution. Therefore, it is not possible to execute it with a pure bytecode to C compiler such as Toba or J2C. Although dynamically loaded classes are interpreted, most of the execution time is spent in the compiled code. Thus, Harissa's generated code is on average 5 times faster than JDK and 3 times faster than Guava.

Other Benchmarks

JHLZip and JHLUnzip tools [31] insert and extract files from an archive. The tested version of these tools does not include compression and therefore, execution is dominated by I/Os. Our tests have been done using the JDK 1.0.2 classes.zip file as input. As it could be expected, compilers (off-line and JIT) achieve the same level of performance. Finally, JDK is only 1.5 slower than the compilers.

The tested implementation of Othello game [28] allocates a finite time to the computer player to solve one move. The depth of the search depends on the speed of the generated code. We give the time spent to solve up to depth 5 on the first move.

5.3 CHA Evaluation

The impact of the class hierarchy analysis has been studied for many object-oriented languages, including Java. It has been shown that this analysis can improve program performance between 23% to 89% [11]. Table 6 presents the impact of CHA for the programs we have benchmarked. It shows that our CHA implementation allows between 14% to 40% of the virtual call points to be transformed into procedure calls.

6 Conclusion and Future Work

The contribution of this work is threefold: (i) we have designed a hybrid environment for Java, named Harissa, that permit mixing

	JDK 1.0.2			*Guava 1.0 beta 4*				*Harissa*			
	real mn	user cpu sec.	sys cpu sec.	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
Jax	0:30	24	2.3	0:27	19	3.2	**1.1**	0:05	2.6	0.9	**6**
Jell	0:37	30	3	0:28	21	3.3	**1.3**	0:08	3.7	1	**4.6**
Jas	0:53	28	3.5	0:26	15	3.8	**2**	0:11	3.4	1.6	**4.8**
Kawa (codegen)	0:34	27	2.7	0:20	14	3.5	**1.7**	0:06	3.5	0.9	**5.6**

Table 4: Javadoc execution time

	JDK 1.0.2			*Guava 1.0 beta 4*			
	real mn	user cpu sec.	sys cpu sec.	real mn	user cpu sec.	sys cpu sec.	ratio JDK
JHLUnzip	1:34	21	7.1	1:00	6	8.3	**1.5**
JHLZip	0:34	20	8.11	0:22	4.7	8.7	**1.5**
Othello	22			1.5			**14.6**

	Toba 1.0.b4				*Harissa*			
	real mn	user cpu sec.	sys cpu sec.	ratio JDK	real mn	user cpu sec.	sys cpu sec.	ratio JDK
JHLUnzip	0:56	4	6.6	**1.7**	0:56	3	6	**1.7**
JHLZip	0:20	5.4	7.2	**1.7**	0:18	2.3	7	**1.8**
Othello	0.85			**26**	0.50			**44**

Table 5: Other Benchmarks

	Javac	*Javadoc*	*JHLZip*	*JHLUnzip*	*Othello*	*Kawa*
Number of classes, interfaces, and arrays	280	281	99	100	103	213
Interfaces	10	10	6	6	4	12
Arrays	25	25	13	13	16	24
Methods containing bytecode	1867	1910	703	696	701	1310
Native methods	104	104	89	89	95	93
Average size of a method (in bytes)	63.8	64.8	48	47.9	41.2	45.6
Virtual call points	4734	4333	863	845	642	2334
Number of optimized calls due to CHA percentage of virtual call points	**1828** **38%**	**1689** **40%**	**155** **18%**	**155** **18%**	**92** **14%**	**690** **25%**

Table 6: Detailed analysis of method and classes

of interpretation with compiled bytecode, (ii) we have designed an aggressive bytecode to C compiler whose generated code is more efficient than other compilers, and (iii) we have measured the relative efficiency of code produced by off-line and JIT compilers.

Tradeoffs Between JIT and Off-line Compilers?

The micro-benchmarks presented in section 5.1 clearly show that an optimized off-line compiler such as Harissa's is faster than a JIT compiler. The gap between JIT and off-line compilers is greater for the SPARC than for the Pentium. This is due to the fact that binary code for modern RISC processors is complex to optimize and requires analyses that are hard to run in the short time allocated to on the fly compilation.

However, the JIT and off-line strategies can be made complementary. As shown by Plezbert and Cytron [7], a compilation process can consist of running the unoptimized code while another process does aggressive compilation on the background. Once the optimized code is available, the unoptimized code is replaced with the optimized one. Since our system already mixes bytecode interpretation and binary execution, this *continuous compilation scheme* can be incorporated easily in Harissa.

Opportunities for Further Optimizations

While Harissa generated code is already fast, our micro-benchmarks show that there are still opportunities for improvement. In a near future, we plan to integrate an analysis for eliminating transient stack variables so as to be independent from Suif. Furthermore, we are studying the development of a transformation phase, based on control flow information, aimed at rebuilding loop constructs. As was shown earlier, structured programs are usually better compiled.

Finally, we also plan to eliminate some type and bound checks since close examination of the C code generated has demonstrated that most of could be evaluated statically by means of a simple intra-procedural analysis.

Acknowledgments

We would like to thank the Compose Group at Irisa and, in particular, Julia Lawall and Renaud Marlet for their helpful suggestions.

Availability

A binary version of our system is freely available by WWW and can be down-loaded from: `http://www.irisa.fr/compose/harissa-/harissa.html`

References

[1] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[2] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[3] OSF. The J-lite project. URL: http://www.gr.osf.org/java/jlite/-index.htm, 1996.

[4] David Intersimone. Interview with Régis Crelier. URL: http:/www.borland.com/internet/java/interviews/-regis2.html, 1996.

[5] Symantec Corporation. Just-in-time compiler for Windows 95/NT. URL: http://cae.symantec.com/cafe/-fs_jit.html, 1996.

[6] Softway. Introduction to Guava. URL: http://guava.softway.com.au/-introduction.html, 1996.

[7] M.P. Plezbert and R.K. Cytron. Does "just in time" = "better late than never". In *Proceedings of POPL'97*, pages 120–131, Paris (France), January 1997.

[8] T. Keishiro. J2c Java .class to C translator. URL: http://www.webity.co.jp/-info/andoh/java/j2c.html, 1995.

[9] T.A. Proebsting, G. Townsend, P. Bridges, J.H. Hartman, T. Newsham, and S.A. Watterson. Toba: Java for applications - a way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[10] Pendragon Software. Caffeinemark 2.5. URL: http://www.webfayre.com-/pendragon/cm2/index.html, 1996.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[12] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *OOPSLA' 96 Conference*, pages 93–100, San Jose (CA), October 1996.

[13] D. Ford. Jive: A Java decompiler. Technical Report RJ 10022, IBM Research Center, Yorktown Heights, May 1996.

[14] H.P. van Vliet. Mocha - Java decompiler. URL: http://www.inter.nl.net/users-/H.P.van.Vliet/mocha.htm, 1996.

[15] B. Case. Java virtual machine should stay virtual. *Microprocessor Report*, pages 14–15, April 1996.

[16] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 94.

[17] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[18] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[19] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[20] F. Yellin. Low level security in Java. URL: http://java.sun.com/sfaq/verifier.html, December 1995.

[21] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *Proceedings of OOPSLA '95*, pages 108–123, Austin, TX, October 1995.

[22] H.J Boehm and M.Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

[23] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5),ACM Press.

[24] C. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report 96-06-02, Department of Computer Science, University of Washington, June 1996.

[25] M. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, Austin, TX, June 1995.

[26] G. Aigner and U. Holzle. Eliminating virtual calls in C++ programs. In *Proceedings of ECOOP '96*, Linz, Austria, August 1996. Springer-Verlag.

[27] U. Holzle and D. Ungar. Optimizing dynammically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language and Design Implementation*, pages 29(6):326–336. SIGPLAN Notices, June 1994.

[28] S. Voges. Othello game. voges@informatik.uni-muenchen.de, 1997.

[29] R.A. Milowski and P. Bothner. The Kawa scheme interpreter project. URL: http://www.copsol.com/kawa/index.html, 1996.

[30] KB Sriram. Free tools for java. URL: http://www.blackdown.org/~kbs/. 1996.

[31] J. Leach. John's java page. URL: http://lenna.easynet.it/~jhl/java.html, 1996.

Montana Smart Pointers: They're Smart, and They're Pointers

Jennifer Hamilton*
Microsoft
jenh@microsoft.com

Abstract: The Montana C++ programming environment provides an API interface to the compiler, which allows the compilation process to be extended through programmer-supplied tools. This paper investigates the feasibility of that interface, using smart pointers as an example. Smart pointers are a powerful feature of the C++ language that enable a variety of applications, such as garbage collection, persistence, and distributed objects. However, while smart pointers can be used in much the same way as built-in pointers, they are not interchangeable. Using the Montana API, smart pointer functionality can be introduced for built-in pointers, thus enabling built-in pointers that act like smart pointers. We provide an overview of the Montana programming environment and describes how smart pointers can be implemented using the Montana API.

1. Introduction

The Montana[1] C++ programming environment is a joint development effort between IBM's Software Solutions and Research Divisions, and will be the base for a future release of IBM's VisualAge C++ product. Montana provides many unique features over traditional C++ compilers, most notably support for complete incremental compilation and an API interface [Nac96].

The purpose of this paper is to assess the feasibility of the Montana API interface for extending the compilation process to augment built-in language syntax. We have chosen the C++ smart pointer support as a basis of comparison. In this paper we present a partial smart pointer implementation using a Montana extension, where built-in pointer operations are modified as part of the compilation process, and summarize the results.

2. The Montana C++ Programming Environment

The Montana project grew from the recognition that current C++ development environments, while improving, were lacking in many areas, especially compared with those available for languages such as Smalltalk. One of the major frustrations in developing large C++ applications is the build turnaround time. The goal for Montana is to provide extremely fast incremental compilation, so that recompilation time required is proportional to the size of the change. In particular, changing a header files should not force recompilation of all files that happen to include it.

The design goal for the Montana architecture is that it can be extended in a variety of ways. A good example is the Montana object model[2] support. Most C++ compilers support a single native object model, the semantics for which are entrenched in the compiler itself, making it difficult to support different object, such as DirectToSOM C++ [Ham96] or other industry object models. Montana, however, was designed so that the object model is supported through a well-defined interface. A new object model can be added without requiring massive changes throughout the compiler. At the time of writing, the author was responsible for the design and development of such non-native object models.

Montana is designed around a system called *CodeStore* [Bar94]. CodeStore consists of a C++ parser, a database that contains the compiled C++ program representation, and a class library that provides an API interface to the compilation process and program representation. Using this class library interface, C++-knowledgeable tools such as browsers can query the program representation of a compiled C++ program. In addition, CodeStore tools called *extensions* can be written that interact with the compilation process.

There are three types of extensions [Sor96]: 1) *CodeStore*

* This work was performed while the author was a member of the C++ compiler development group at the IBM Toronto Laboratory.

[1] The name "Montana" originated from an architecture meeting in which the idea of developing a new compiler with a clean slate was referred to as a "blue sky" approach. Since "blue sky" was thought to be the motto for the state of Montana (it's actually "big sky"), that became the name of the project. [Nac96]

[2] By object model, we mean issues such as how objects are laid out in memory and the strategy used to support virtual functions and bases. See [Lip96] for a detailed discussion.

extensions, which add data to the CodeStore and have incremental update capability, 2) *incorporation*[3] *extensions*, that modify or observe the incorporation process directly and 3) *user interface extensions*, which allow additional artifacts such as buttons and menus to be added to the user interface display. An example of the first type of extension is a separate compiler that is triggered as part of the compilation process to handle different file types, while an example of an incorporation extension is a tool that interacts directly with the compilation process itself, querying or updating the result. In this paper, we will concentrate on the second form of extension.

3. Smart Pointers

Smart pointers are a powerful feature of the C++ language that enable a variety of applications, such as garbage collection, persistence, and distributed objects. They are used to augment the functionality of C++ pointer operations, allowing the programmer to perform additional work when pointers are created and used.

Smart pointers essentially allow a user-defined exit added to be pointer operations. A smart pointer [Stro89] itself is an instance of a class that wraps a built-in pointer, for which the dereference operator -> has been overloaded, as shown in Figure 1 (smart pointers are typically defined using templates however). Such objects can be used in much the same way as a built-in pointer, but have additional functionality provided through operator overloading. In much the same way as inheritance, smart pointers can be used in C++ to extend the functionality of a class. However, while inheritance extends the functionality of class instances themselves, smart pointers are used to extend the environment containing the instance. In other words, smart pointers are used to modify how the programming environment operates on an object, rather than how the object operates on itself. Smart pointers have a wide variety of uses, from simple applications such as detecting null dereferences, debugging, and read-only pointers [Alg95], to more complex applications such as garbage collection [GC96], [Ede92a], and persistence [Coh96].

```
#include <iostream.h>

struct S {
    int i;
};

class SP {
    S *_p;
  public:
    SP(S *p) : _p(p) {}
    S* operator->() {
        cout << "dereferencing" << endl;
        return _p;
    }
};

int main()
{
    SP sp(new S);

    sp->i = 10;        // sp.operator->()->i
}
```

Figure 1 Simple Smart Pointer Class

In general, smart pointers can be used in exactly the same way as built-in pointers, however, as described in [Ede92b], there are some important differences between the two with respect to implicit type conversions performed by the compiler. These fall into two major categories: 1) class hierarchies and 2) types qualified with `const` or `volatile`. A further issue, described in [Mey96a] and [Mey96b], is testing for nullness. When using built-in pointers, the compiler implicitly performs a variety of conversions between pointer types. Examples are `T*` to `const T*`, `Derived*` to `Base*`, and `T*` to `void*`. These implicit conversions are not supported for smart pointer types.

If, however, the "smarts" of a smart pointer could be added to a built-in pointer, these problems would be alleviated. In this section, we describe the changes that would be needed to built-in pointer expressions in order that they operate as smart pointers. For the purpose of this example, we will implement a reference counting smart pointer. In subsequent sections, we will describe how to implement this model using a Montana incorporation extension.

3.1 A Reference Counting Smart Pointer

The basic model for a reference-counting smart pointer is as follows:

1) Whenever a new reference is made to a given object, the reference count for that object should be incremented.
2) If a reference to an object is removed, the reference count for that object should be decremented. If the reference count for an object goes to zero, delete the object.

These rules are illustrated by the functions `increment`

[3] *Incorporation* is the Montana term for recompiling a program, in which the changes to the source will be incorporated into the CodeStore database.

```
void decrement(ReferenceCounter *sp)
{
    if (!sp)
        return;
    if (! --sp->rc)
        delete sp;
}

void increment(ReferenceCounter *sp)
{
    if (!sp)
        return;
    ++sp->rc;
}
```

Figure 2 Reference Counter Functions

and `decrement` shown in Figure 2.

In order to add reference counting smart pointer functionality to built-in pointers operations, the following expression transformations are required:

Pointer assignment: Whenever an assignment is made to a designated smart built-in pointer, the reference count for the object originally pointed to should be decremented and that of the object now pointed to should be incremented. Thus, the expression `p1 = p2` becomes:

```
(p1 == p2 ? 0 : decrement(p1),
   p1 = p2, increment(p1), p1)
```

Pointer initialization: A designated smart built-in pointer must always be either explicitly initialized to a value, or to zero. (If a pointer were not initialized to zero and contained non-zero garbage, a subsequent assignment to that pointer using the previous expression would likely result in an exception).

The statement `SPC* p1;` becomes:

```
SPC* p1 = 0;
```

and `SPC** p1 = new SPC*;` becomes:

```
SPC** p1=new SPC*; p1 ? *p1=0 : 0;
```

If a smart pointer is initialized to a value, the reference count for the underlying object must be incremented. So the statement `SPC* p1 = p2;` becomes:

```
SPC* p1 = p2; increment(p1);
```

Object Initialization: When a designated smart built-in pointer object is created, the reference count must be initialized. For dynamically-created objects, the count should be initialized to 0, and for static or automatic objects, the reference count should be initialized to 1 so that the object can be used in reference counting contexts, but will never be deleted.

Pointer destruction: When a designated smart built-in pointer is destroyed the reference count for the referenced object must be decremented. There are several ways that a smart pointer will be destroyed, the most common being that it goes out of scope. Other possibilities are that a dynamically allocated smart pointer is deleted, or an exception occurs in which the containing block is unwound from the stack. Only the deletion of a dynamically-allocated smart pointer consists of an expression that can be transformed. The other two require modifications to the function itself so that the scope termination and exception handling code will include the decrement of any smart pointers declared therein.

3.2 Which Built-in Pointers Become Smart?

The above discussion raises the question of how to determine which built-in pointer operations should be transformed into smart pointer operations. One could blindly apply the transformation to all built-in pointer operations, but this would certainly be overkill. Rather, we would like to select only specific pointers for the transformation The approach that we have chosen is to define a special base class, `ReferenceCounter`. Expressions involving objects declared of, or pointers to, a class derived from `ReferenceCounter` will be transformed as described above.

For example, consider the built-in pointers declared of type `C*` in Figure 3. Because class `C` is derived from `ReferenceCounter`, several transformations should take place. `cp1` and `cp2` should be implicitly initialized to 0 at the point of declaration, and the assignment from `cp2` to `cp1` should be transformed as described earlier.

```
class C : public ReferenceCounter {};

int main()
{
    C *cp1, *cp2;

    cp1 = cp2;
}
```

Figure 3 Built-in Pointer Operations

4. Montana CodeStore Architecture

The previous section described the necessary

transformations to built-in pointer operations in order to implement a reference counting smart pointer. This section provides an overview of the Montana CodeStore architecture and describes in a generic sense how a transformation extension can be added. This will form the basis for the remainder of the paper, which describes our implementation of smart built-in pointers in Montana using an incorporation extension.

4.1 The Montana Incorporation Process

As part of the Montana incremental compilation process, the compiler separates a source file into *regions*, where each region consists of approximately one declaration. If a region, or something that region depends upon, has changed since the last incorporation, it is re-incorporated. Re-incorporation involves a number of standard steps: parsing, semantic analysis, transformation, error checking, code generation, and incremental linking. In addition, dependency arcs are added between CodeStore elements so that a change in one region can trigger a re-incorporation of a dependent region. For example, a region containing a derived class declaration will have a dependency on each region containing one of its base classes.

The Montana class `CS_CodeStore` is used to represent the underlying CodeStore database. This class supports a variety of routines to create, query and update the CodeStore. An application that operates on a CodeStore will contain exactly one instance of the `CS_CodeStore` class. If an incorporation is currently taking place against the database, this `CS_CodeStore` instance will contain a reference to an object of type `CS_IncorporationState`, which represents the current state of the incorporation.

4.2 Transformation

The transformation step involves simplifying expressions and statements into a C-like representation. In Montana, it is implemented through the class `CS_Transformer`, which is shown in Figure 4. `CS_Transformer` has three versions of the method `transform`, corresponding to the different types of transformations that are supported. Most calls to the `CS_Transformer::transform` methods are for statements or initializations. Expression transformations typically occur as part of the transformation of their containing statement.

When the compiler needs to transform an item, it obtains a transformer object from the CodeStore's incorporation state object. The incorporation state in turn retrieves the transformer from an implementation component factory (see Figure 5). The incorporation state maintains a list of implementation component factories, and selects the transformer returned by the front element in the list.

An implementation component can be one of four types: a type analyzer, a diagnostician, a transformer, or an optimizer. Each of these implementation components take part in a specific portion of the incorporation process, and can be overridden to modify the compilation process. Applications can provide custom implementation components by subclassing the implementation component factory class and providing overrides for the methods of interest. By inserting this new class at the front of the incorporation state list, the incorporation state will select the overridden component provided.

For example, to provide a transformer (incorporation) extension, the class `CS_ImplementationComponentFactory` would be derived from, supplying a transformer method that would return the custom transformer object. The incorporation state method `prependImplementationComponentFactory` would be called to add this factory to the front of the list. Any factory methods that are not overridden would return the result of invoking that method against the next factory in the list, as shown in Figure 5 with the method invocation against the result of the `next` method.

Montana supplies a default implementation component factory that provides the standard implementations for each component. When no extensions have been introduced, this factory will be at the front of the incorporation state's list. Figure 6 shows the relationship between the various classes discussed in this section.

5. Implementing Smart Pointers with Montana

In this section, we will present our implementation of smart pointers through built-in pointers using a Montana transformation incorporation extension. The complete implementation is included in the Appendix A, and we have extracted specific pieces to clarify the explanation. We will first describe how to add our specific transformation extension, and then present our implementation for reference counting smart pointers based on the required expression transformations discussed earlier.

```
class CS_Transformer : public CS_IncorporationComponentBase<CS_DepthFirstModifier>
{
public:
   CS_Transformer(CS_IncorporationState& s) :
CS_IncorporationComponentBase<CS_DepthFirstModifier>(s) { }

   // Transform a statement tree
   //
   virtual CS_bool transform(CS_Statement*& stmt, CS_bool emitMessages)
      { modifyStatement(*stmt); return CS_true; }

   // Transform a variable initializer
   //
   virtual CS_bool
         transform(CS_Initializer*& init, CS_VariableDeclaration& var, CS_bool emitMessages)
      { init = &modifyInitializer(*init, &var.typeDescriptor(), &var); return CS_true; }

   // Transform an expression tree
   //
   virtual CS_Expression& transform(CS_Expression& expr, CS_bool emitMessages)
      { return modifyExpression(expr); }
};
```

Figure 4 `CS_Transformer`

```
class CS_ImplementationComponentFactory : public CS_Link<CS_ImplementationComponentFactory>
{
public:
   virtual CS_TypeAnalyzer&   typeAnalyzer()   { return next()->typeAnalyzer();   }
   virtual CS_Diagnostician&  diagnostician()  { return next()->diagnostician();  }
   virtual CS_Optimizer&      optimizer()      { return next()->optimizer();      }
   virtual CS_Transformer&    transformer()    { return next()->transformer();    }
};
```

Figure 5 `CS_ImplementationComponentFactory` class

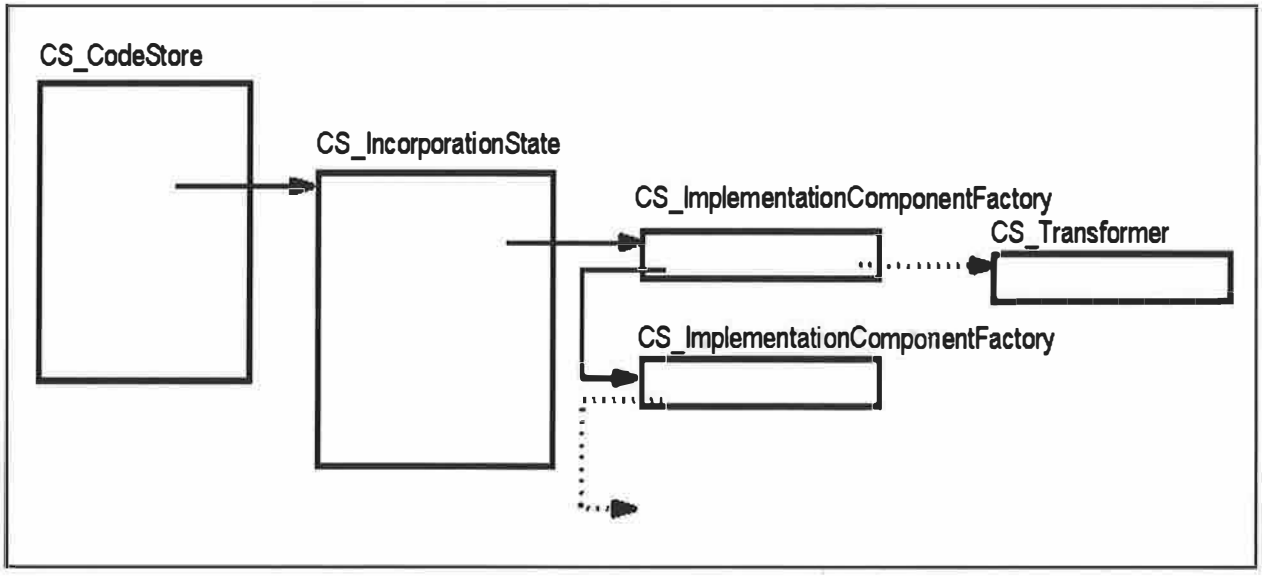


Figure 6 Relationship Between Classes

```
SmartPointerImplementationComponentFactory::
        SmartPointerImplementationComponentFactory(CS_IncorporationState& s) :
    _state(s),
    // The second argument to this constructor comes from pull on the
    // chain of components stored in the IncorporationState.
    //
    _transformer(new SmartPointerTransformer(_state,
                _state.implementationComponentFactory().transformer()))
{
    assume(_transformer);
}

CS_Transformer& SmartPointerImplementationComponentFactory::
        transformer()
{
    assume(_transformer);
    return *_transformer;
}

SmartPointerImplementationComponentFactory::
        ~SmartPointerImplementationComponentFactory()
{
    delete _transformer;
}
```

Figure 7 SmartPointerImplementationComponentFactory implementation

```
class SmartPointerImplementationComponentFactory
   : public CS_ImplementationComponentFactory {
  public:
    SmartPointerImplementationComponentFactory(CS_IncorporationState&);
    virtual CS_Transformer& transformer();
    virtual ~SmartPointerImplementationComponentFactory();

  private:
    CS_IncorporationState& _state;
    CS_Transformer* _transformer;
};
```

Figure 8 `SmartPointerImplementationComponentFactory` class

5.1 Creating a Transformation Incorporation Extension

As described in the previous section, in order to implement a transformation extension, we must create a subclass of the `CS_ImplementationComponent-Factory` class and insert an object of this new type at the front of the incorporation state's factory list. Figure 8 shows the definition of the class `SmartPointerImplementationComponent-Factory` and Figure 7 shows the corresponding implementation. The method transformer is overridden to return our custom smart pointer transformer extension. The constructor for the factory initializes the `_transformer` member by creating a new object of class `SmartPointerTransformer`. This latter class will implement our smart pointer transformer extension, and will be discussed in more detail subsequently. Note that the transformer extension constructor is passed the incorporation state and the current transformer object, obtained from the front of the factory list.

5.2 Dynamically Loading a Transformation Incorporation Extension

We now have a factory implementation that will return our custom transformation extension. The next issue to address is how the factory object will be created and added to the front of the incorporation state's factory list. This will be achieved by loading a dynamic link library (DLL) that contains a static variable whose initialization will cause the factory to be created and inserted into the list. Then the question is, how is the DLL loaded? We will now examine the Montana support for defining and loading extensions.

Externally, a Montana extension is introduced using an *Incremental C++ Extension*, or *ice*, file. Montana searches for and applies ice files at load time according to a defined search order. Figure 9 shows an ice file which defines an extension called `SmartPointer`, for which the corresponding DLL to load is `smartp.dll`. The suffix and prefix information is used to associate a

specific file type with a given extension. This interface is for CodeStore extensions, but is used as a temporary measure until the final interface for incorporation extensions is defined.

Montana programs are compiled by providing a *configuration file* which supplies the various options for the compilation. For our purposes, the configuration file shown in Figure 10 is used.

This configuration file indicates that the source file to be compiled is `t.cpp`, the target executable will be `t.exe`, and that an additional source file called `dummy.sp` will also be processed. This latter source file, being an unsupported file type, will cause Montana to search the ice files for an appropriate extension that handles this file type, and load the extension DLL `smartp.dll`.

The next step is to register an *extension dynamic load point* using a statically-defined variable in the `smartp.dll` extension DLL as shown in Figure 11. An extension dynamic load point is used to register an extension with the compiler. For an incorporation extension, the final parameter to the extension dynamic load point constructor, the *incorporation startup function pointer* is most important. This function will be run at the start of every incorporation and can be used by an extension to plug in components into the incorporation state.

```
[SmartPointer]

type=extension
description=Smart Pointer
Extension
dll=smartp.dll
suffixes=sp SP
prefix=dummy
```

Figure 9 ice File for Smart Pointer Extension

```
source type(cpp) src0 = "t.cpp"
target "t.exe" { source src0 }

source type(sp) src1 = "dummy.sp"
```

Figure 10 Montana Configuration File

```
CS_ExtensionDynamicLoadPoint

SmartPointer::extension_load_point(
        SmartPointer::className(),
        SmartPointer::update,
        SmartPointer::isChanged,

SmartPointer::processOptions,
        EXTENSION_PRIORITY,

SmartPointer::incorporationStartup);
```

Figure 11 Extension Dynamic Load Point

```
class SmartPointer : public CS_InterfaceBase
{
public:
   static const char* className();

   static void SmartPointer::incorporationStartup(
      CS_ExtensionDynamicLoadPointLink&, CS_IncorporationState&);

   static CS_DependencyNode::UpdateResult
        update(CS_ExtensionSource* me, CS_IncorporationState& state,
               CS_bool emitMessages);

   static CS_bool isChanged(CS_ExtensionSource* me);

   static void processOptions(CS_ExtensionSource* me, CS_OptionList& options);

private:
   static CS_ExtensionDynamicLoadPoint  extension_load_point;
};
```

Figure 12 `SmartPointer` class

```
void SmartPointer::incorporationStartup(
        CS_ExtensionDynamicLoadPointLink&, CS_IncorporationState& state)
{
    cout << __FUNCTION__ << endl;

    SmartPointerImplementationComponentFactory* fac =
        new SmartPointerImplementationComponentFactory(state);
    assume(fac);  // (our version of "assert")

    // Push our new factory with its new Transformer onto the chain
    // stored in the IncorporationState.
    //
    state.prependImplementationComponentFactory(*fac);

    return;
}
```

Figure 13 `incorporationStartup` method

The main effect then, of constructing the static member variable `SmartPointer::extension_load_point` is that the method `SmartPointer::incorporationStartup` will be called prior to each incorporation. The class `SmartPointer` and the `incorporationStartup` method are shown in Figure 12 and Figure 13. In the `incorporationStartup` method the newly-created `SmartPointerImplementationComponentFactory` object is added to the front of the incorporation state's factory list (see Figure 6). Adding this custom factory object to the front of the queue will cause any requests made of the incorporation state for a transformer object to return the custom transformer, `SmartPointerTransformer`.

5.3 The SmartPointerTransformer class

At this point, whenever a transformation takes place, the `SmartPointerTransformer` class (see Figure 14) will have control. One of the three overridden `transform` methods shown at the beginning of the class will be called depending upon the type of transformation taking place: a statement, initialization, or an expression. The overridden versions of the `SmartPointerTransformer` methods are shown in Figure 15. These `transform` methods have fairly standard implementations. They first call an appropriate `modify` method, and then invoke the `transform` method of the previous element in the component chain (given by member variable `_parent`.)

Recall that the constructor for the `SmartPointerTransformer` class is passed the current transformer object, which is used to initialize the data member `_parent`. Calling the parent `transform` method allows the standard compiler transformations to take place after the extension has been run.

It is when the `modify` method is called that the transformer extension has an opportunity to modify the transformed expression. The compiler-supplied `modify` methods step through the underlying item and calls an appropriate `modifyxxx` method for each entity encountered. By overriding methods corresponding to expression of interest, the transformer extension can modify these expressions. In this case, we have overloaded `modifyAssignExpression`, `modifyExpressionInitializer`, `modifyImplicitInitializer`, and `modifyDestructorStateChangeExpression` (these methods will be explained in more detail in the next section). If the underlying expression or statement corresponds to one of these four, the overloaded method will be called. Each of these methods determines if any further expression transformation is necessary, based on the type of the object being operated on. If so, the expression is transformed according to the model described earlier for transforming built-in pointer operations into smart pointer operations.

5.4 CS_SmartPointerTransformer::modify

Now we will discuss the implementation of the `CS_SmartPointerTransformer::modify` methods. Each of these methods uses the `SmartPointerTransformer::transformerImplementation` method to determine if the current expression or statement deals with an object of interest. This method returns a `SmartPointerTransformationImplementation` that will perform implementation-specific transformations, depending upon the smart pointer type. We will discuss this latter class in the next section.

```
class SmartPointerTransformer : public CS_Transformer {
  public:
    SmartPointerTransformer(CS_IncorporationState& s, CS_Transformer& p)
        : CS_Transformer(s), _parent(p),
          _referenceCounterTransformerImplementation(0) {
    }
    ~SmartPointerTransformer();

    virtual CS_bool transform(CS_Statement*& stmt, CS_bool emitMessages);
    virtual CS_bool transform(CS_Initializer*&, CS_VariableDeclaration&, CS_bool);
    virtual CS_Expression& transform(CS_Expression&, CS_bool);

    virtual CS_Expression& modifyAssignExpression(CS_BinaryExpression&);
    virtual CS_Initializer& modifyExpressionInitializer(
        CS_ExpressionInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);
    virtual CS_Initializer& modifyImplicitInitializer(
        CS_ImplicitInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);
    virtual CS_Expression& modifyDestructorStateChangeExpression(
        CS_DestructorStateChangeExpression&);

    CS_Expression& typeAnalyze(CS_Expression&);
    CS_Initializer& typeAnalyze(CS_Initializer&);

  private:
    CS_Transformer& _parent;

    // classes for each smart pointer implementation
    ReferenceCounterTransformerImplementation* _referenceCounterTransformerImplementation;

    // Return the smart pointer implementation, if any, for the expression
    // The expression must be a pointer to a class derived from a SmartPointer class
    SmartPointerTransformerImplementation* transformerImplementation(CS_TypeDescriptor&);
};
```

Figure 14 `SmartPointerTransformer` class

```
CS_bool SmartPointerTransformer::transform(CS_Statement*& stmt, CS_bool emitMessages)
{
    modifyStatement(*stmt);
    _parent.transform(stmt, emitMessages);
    return CS_true;
}

CS_bool SmartPointerTransformer::
        transform(CS_Initializer*& init, CS_VariableDeclaration& var, CS_bool emitMessages)
{
    init = &modifyInitializer(*init, &var.typeDescriptor(), &var);
    _parent.transform(init, var, emitMessages);
    return CS_true;
}

CS_Expression& SmartPointerTransformer::transform(CS_Expression& expr, CS_bool emitMessages)
{
    CS_Expression *expr2 = &modifyExpression(expr);
    return _parent.transform(*expr2, emitMessages);
}
```

Figure 15 `SmartPointerTransformer transform` methods

```
CS_Expression& SmartPointerTransformer::modifyAssignExpression(CS_BinaryExpression& binary)
{
    if (! binary.expression1().typeDescriptor().isPointer())
        return binary;

    SmartPointerTransformerImplementation *ti =
        transformerImplementation(*binary.expression1().typeDescriptor().next());

    return ti ? typeAnalyze(ti->modifyAssignExpression(binary)) : binary;
}
```

Figure 16 `modifyAssignExpression` method

```
CS_Initializer& SmartPointerTransformer::modifyExpressionInitializer(
        CS_ExpressionInitializer& init, CS_TypeDescriptor* td, CS_VariableDeclaration* var)
{
    if (!td)
        return init;

    // need to handle both pointer and object initialization
    SmartPointerTransformerImplementation *ti =
        transformerImplementation(td->isPointer() ? *td->next() : *td);

    if (! ti)
        return init;

    return typeAnalyze(ti->modifyExpressionInitializer(init, td, var));
}
```

Figure 17 `modifyExpressionInitializer` method

```
CS_Initializer& SmartPointerTransformer::modifyImplicitInitializer(
        CS_ImplicitInitializer& init, CS_TypeDescriptor* td, CS_VariableDeclaration* var)
{
    if (!td)
        return init;

    // need to handle both pointer and object initialization
    SmartPointerTransformerImplementation *ti =
        transformerImplementation(td->isPointer() ? *td->next() : *td);

    if (! ti)
        return init;

    return typeAnalyze(ti->modifyImplicitInitializer(init, td, var));
}
```

Figure 18 `modifyImplicitInitializer` method

```
CS_Expression& SmartPointerTransformer::
        modifyDestructorStateChangeExpression(CS_DestructorStateChangeExpression& dsce)
{
    return SmartPointerTransformerImplementation::
        modifyDestructorStateChangeExpression(dsce);
}

CS_Expression& SmartPointerTransformerImplementation::
    modifyDestructorStateChangeExpression(CS_DestructorStateChangeExpression& dsce)
{
    // save most recent state table entry
    if (dsce.tableEntry() && dsce.tableEntry()->asDestructorStateTableEntry())
        _currentDestructorStateTableEntry = dsce.tableEntry()->asDestructorStateTableEntry();

    return dsce;
}
```

Figure 19 `modifyDestructorStateChangeExpression` methods

The `modifyAssignExpression` method is shown in Figure 16. For our smart pointer implementation, we want to detect any pointers assignments where the underlying type is derived from a special base class such as `ReferenceCounter`. `modifyAssignExpression` is passed a reference to a `CS_BinaryExpression` object, representing the assignment expression currently being transformed. If the type descriptor for the lhs of this expression, given by `CS_BinaryExpression::expression1()`, is not a pointer, then no further work is necessary. If it is a pointer, then the type to which it points, given by `CS_TypeDescriptor::next()` is passed to `transformerImplementation` to check if it points to an object derived from one of the special base classes. If a non-null value is returned, the `modifyAssignExpression` of the returned implementation object is called to modify the expression.

The `modifyExpressionInitializer` and `modifyImplicitInitializer` methods (Figure 17 and Figure 18) perform similar functions, but must handle both pointer and non-pointer initializations. These methods call the `transformerImplementation` with the `CS_TypeDescriptor` for the non-pointer variable being initialized. If the variable being initialized is a pointer, the `CS_TypeDescriptor` for the type pointed to, given by `td->next()`, is passed instead.

The `modifyDestructorStateChange-Expression` method, shown in Figure 19, does not actually perform any modification against the given expression. Rather, it calls the static method `SmartPointerTransformationImplementation::modifyDestructorStateChangeExpression`, which simply saves the most recently seen state table entry if that entry is for a destructor. This value will be used later to add information to the state table in the appropriate location.

5.5 SmartPointerTransformerImplementation class

As discussed earlier, the model for our smart pointer implementation is that a built-in pointer will be transformed into a smart pointer if the underlying type inherits from a special base class. In order to provide multiple smart pointer transformations, we have defined a common base class called `SmartPointer-TransformerImplementation` (Figure 20), from which a derived class will be defined for each smart pointer implementation. This derived class will handle the transformations specific to that smart pointer implementation.

When a method needs to determine if a smart pointer implementation applies to a given expression, it calls the method `SmartPointerTransformer::transformerImplementation`, shown in Figure 21. The `transformerImplementation` method is passed a reference to an object of type `CS_TypeDescriptor`, which describes the type of the object being operated on. If the object is of a class type, a reference to the associated `CS_ClassDeclaration` is assigned to the variable `decl`. A `CS_ClassDeclaration` provides complete information about a class declaration, such as the class name, members. etc. So at this point, `decl` will reference the class declaration for the object of interest.

Next, the `findClassDeclaration` method of the `ReferenceCounterTransformerImplementation` class is invoked. This method returns a pointer to the `CS_ClassDeclaration` object representing the class `ReferenceCounter`, if that declaration has been encountered in the program, and null otherwise. If non-null is returned, the method uses the `CS_ANSI_Queries::isBaseClassOf` method to determine if the class declaration for the object of interest is derived from `ReferenceCounter`. The `CS_ANSI_Queries` class provides a variety of functions that support querying of class declarations. If `ReferenceCounter` is a base class, the `_referenceCounterTransformer-Implementation` data member will be initialized with a new `ReferenceCounterTranformer-Implementation` object if it has not yet been initialized.

In the current implementation, we have defined one special base class, `ReferenceCounter`, and one corresponding specialization of `SmartPointerTransformerImplementation`. The programmer would include the declaration of `ReferenceCounter` class (see Figure 22) and derive from it to introduce reference-counting functionality for pointer operations against objects of that derived class. (The `name` member in `ReferenceCounter` is used for debugging purposes and will be discussed later). Additional smart pointer implementations could be introduced by inserting code at the end of the `transformerImplementation` method where indicated.

5.6 ReferenceCounterTransformerImplementation class

The ReferenceCounterTranformerImplementation (see Figure 23) class provides the transformer extension implementation for a reference counting smart pointer. It is a specialization of the `SmartPointer-TransformerImplementation`, and contains overrides of the `SmartPointerTransformer-Implementation::modify` methods, along with additional methods specific to implementing a reference counting smart pointer.

```
class SmartPointerTransformerImplementation : public
CS_IncorporationComponentBase<CS_InterfaceBase> {
  public:
    SmartPointerTransformerImplementation(CS_IncorporationState &state,
                                          SmartPointerTransformer &transformer) :
        CS_IncorporationComponentBase<CS_InterfaceBase>(state), _transformer(transformer) {}

    virtual CS_Expression& modifyAssignExpression(CS_BinaryExpression&) = 0;
    virtual CS_Initializer& modifyExpressionInitializer(
        CS_ExpressionInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*) = 0;
    virtual CS_Initializer& modifyImplicitInitializer(
        CS_ImplicitInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*) = 0;

    static CS_Expression& modifyDestructorStateChangeExpression(
        CS_DestructorStateChangeExpression& dsce);
    CS_DestructorStateTableEntry* currentDestructorStateTableEntry()
        { return _currentDestructorStateTableEntry; }
    void currentDestructorStateTableEntry(CS_DestructorStateTableEntry *ste)
        { _currentDestructorStateTableEntry=ste; }

    SmartPointerTransformer& transformer() { return _transformer; }

  private:

    SmartPointerTransformer& _transformer;
    static CS_DestructorStateTableEntry *_currentDestructorStateTableEntry;
};
```

Figure 20 SmartPointerTransformerImplementation class

```
SmartPointerTransformerImplementation *SmartPointerTransformer::
        transformerImplementation(CS_TypeDescriptor& td)
{
    if (! td.isNamedType() ||
            ! td.declaration().declarationKind() == CS_Declaration::IsClass)
        return NULL;
    CS_ClassDeclaration &decl = *td.declaration().asClassDeclaration();

    CS_ClassDeclaration *referenceCounter =
        ReferenceCounterTransformerImplementation(state(), *this).findClassDeclaration();
    if (referenceCounter &&
            CS_ANSI_Queries::isBaseClassOf(*referenceCounter, decl)) {
        cout << "got a pointer to class derived from "
             << referenceCounter->signature() << endl;
        if (! _referenceCounterTransformerImplementation) {
            _referenceCounterTransformerImplementation =
                new ReferenceCounterTransformerImplementation(state(), *this);
        }
        return _referenceCounterTransformerImplementation;
    }
    // insert code to look for other smart pointer class implementations here
    return NULL;
}
```

Figure 21 `transformerImplementation` method

```
class ReferenceCounter {
  private:
    int rc;
    char *_name;

    static void dtor(ReferenceCounter **sp, int);
    static void decrement(ReferenceCounter *sp);
    static void increment(ReferenceCounter *sp);

  public:
    ReferenceCounter(char *name) : _name(name) { rc = 0; }
    char *name() { return _name; }
    virtual ~ReferenceCounter();
};
```

Figure 22 ReferenceCounter class

```
class ReferenceCounterTransformerImplementation :
        public SmartPointerTransformerImplementation {
  public:
    ReferenceCounterTransformerImplementation(
        CS_IncorporationState& state, SmartPointerTransformer& transformer) :
        SmartPointerTransformerImplementation(state, transformer) {};

    CS_ClassDeclaration* findClassDeclaration();

    CS_Expression& modifyAssignExpression(CS_BinaryExpression& binary);
    CS_Initializer& modifyExpressionInitializer(
        CS_ExpressionInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);
    CS_Initializer& modifyImplicitInitializer(
        CS_ImplicitInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);

    private:
        const CS_Atom& referenceCountMember();
        virtual CS_Expression& decrementReferenceCounterExpression(CS_Expression &);
        virtual CS_Expression& incrementReferenceCounterExpression(CS_Expression &);
        virtual CS_Expression& createStateChangeExpression(
            CS_Expression&, CS_VariableDeclaration&);
        virtual CS_FunctionDeclaration& findOrCreateDecrement();
        virtual CS_FunctionDeclaration& findOrCreateIncrement();
        virtual CS_FunctionDeclaration& findOrCreateDtor();
        virtual CS_FunctionDeclaration& findOrCreateMemberFunction(char *);
        virtual CS_DestructorStateTableEntry&createDestructorStateTableEntry(
            CS_VariableDeclaration&, CS_TreeNode&);
        virtual void addDestructorCalls(
            CS_VariableDeclaration&, CS_DestructorStateTableEntry&, CS_TokenLocation&);
};
```

Figure 23 ReferenceCounterTransformerImplementation class

```
CS_Expression&
ReferenceCounterTransformerImplementation::modifyAssignExpression(
    CS_BinaryExpression &binary)
{
    // Don't transform expressions on temporaries
    CS_Expression &e = binary.expression1();
    if (e.expressionKind() == CS_Expression::IsName &&
            e.asNameExpression()->name().declaration() &&
            ! e.asNameExpression()->name().declaration()->mapsToASourceLocation()) {
        return binary;
    }

    CS_TokenLocation loc =
        binary.sourceLocation().sourceRegion()->tokenLocation();

    CS_Expression &expr =
        ef().createCommaExpression(loc,
            decrementReferenceCounterExpression(binary.expression1()),
            ef().createCommaExpression(loc,
                binary,
                incrementReferenceCounterExpression(binary.expression1())));
    return expr;
}
```

Figure 24 modifyAssignExpression method

```
CS_Initializer& ReferenceCounterTransformerImplementation::
        modifyExpressionInitializer(CS_ExpressionInitializer& init,
                                    CS_TypeDescriptor* td, CS_VariableDeclaration* var)
{
    assume(var);

    CS_TokenLocation loc =
        init.expression().sourceLocation().sourceRegion()->tokenLocation();

    if (td->isPointer()) {
        CS_BinaryExpression *be = init.expression().asBinaryExpression();
        assume(be);
        // C *c1(x) will already be transformed to C *c1 = x;
        assume(be->binaryExpressionKind() == CS_BinaryExpression::opAssign);

        CS_Expression &expr1 = be->expression1();

        // change C *c1; c1 = x to
        //        C *c1; c1 = (c1=x, c1 != 0 ? c1->rc++: 0, c1)

        init.setExpression(
            ef().createAssignExpression(loc,
                expr1,
            ef().createCommaExpression(loc,
                ef().createAssignExpression(loc,
                    ic().cloneExpression(expr1),
                    transformer().modifyExpression(be->expression2())),
                ef().createCommaExpression(loc,
                    incrementReferenceCounterExpression(
                        expr1),
                    ef().createCommaExpression(loc,
                        createStateChangeExpression(expr1, *var),
                    ic().cloneExpression(expr1))))));
        return init;
    }
    // For non-dynamic variables, initialize reference count
    // to 1 so that never get collected.

    init.setExpression(
        ef().createCommaExpression(loc,
            transformer().modifyExpression(init.expression()),
            ef().createAssignExpression(loc,
                ef().createDotExpression(loc,
                    ef().createNameExpression(loc, *var),
                    referenceCountMember()),
                ef().createLiteralExpression(
                    cs(), loc, intType(), 1))));
    return init;
}
```

Figure 25 modifyExpressionInitializer method

5.6.1 Transforming Pointer Assignments

As discussed earlier, when an assignment to a reference-counting smart pointer occurs, we want to transform an expression such as `x = y`, where x points to a type derived from `ReferenceCounter`, to the following:

```
(x == y ? 0 :
decrement(x), x = y,
increment(x), x)
```

This is achieved through the `modifyAssignExpression` method shown in Figure 24. The first thing the method does is check if the left-hand-side (lhs) of the expression (given by `binary.expression1()`), is a compiler-generated temporary. Unlike programmer-declared variables, the declaration of a temporary will not have a corresponding source location. Our implementation does not currently handle temporaries, and assignments to such are ignored. Temporaries are discussed in more detail in section 7.

If the target of the assignment is not a temporary, a new comma expression is created using the result of the `decrementReferenceCounter` and `incrementReferenceCounter` methods along with

the current assignment expression. The `findOrCreateDecrement` method called in the `decrementRefereneCounterExpression` method locates or creates a declaration corresponding to the `ReferenceCounter::increment` method declared earlier. Note that the right-hand-side (rhs) value is used three times in the resulting expression, in decrement, assignment, and increment expressions. We should generate a temporary to hold the rhs value so that expressions containing side-effects are not executed multiple times. Further, we currently do not generate code to handle the initial check for the lhs being equal to the rhs, which requires a temporary for both the lhs and the rhs, or the final expression containing just the rhs for the assignment value. This support would be added when temporaries are handled by our implementation (see section 7).

5.6.2 Transforming Initialization Expressions

The `modifyExpressionInitializer` method, shown in Figure 25, transforms initialization expressions corresponding to the model described earlier. Much of this method is fairly self-explanatory. For pointers, however, there is an additional action performed, which is to add state change information. If a pointer goes out of scope, either due to an exception or control implicitly returning from the function, the appropriate reference count decrement must take place. This is achieved through calling the `createStateChangeExpression` method, which will use the saved state change variable to create a state change node and insert it in the table in the appropriate place, based on the most recent state change that occurred within the function. This will cause code to be inserted at the end of the function on implicit scope termination to call the `dtor` method defined for the class `ReferenceCounter`, which will decrement the reference count. See the appendix for details.

6. An Example

To demonstrate the smart pointer transformer extension in action, Figure 26 shows a simple program containing several pointer declarations and assignments. Figure 27 shows the resulting execution output after compiling the program with the transformation extension. The built-in pointers act like smart pointers!

```
#include "ReferenceCounterInterface.h"

class C : public ReferenceCounter {
  public:
    int i;
    C(char *name) : ReferenceCounter(name) {}
};

int main()
{
    cout << "C c1;" << endl;
    C c1("c1");

    cout << "C *cp1 = &c1;" << endl;
    C *cp1 = &c1;

    cout << "C *cp2 = new C(\"new C 1\");" << endl;
    C *cp2 = new C("new C 1");

    cout << "cp2 = 0;" << endl;
    cp2 = 0;

    cout << "cp1 = cp2;" << endl;
    cp1 = cp2;

    cout << "C *cp3 = new C(\"new C 2\");" << endl;
    C *cp3 = new C("new C 2");

    cout << "C c2;" << endl;
    C c2("c2");

    cout << "cp1 = &c2;" << endl;
    cp1 = &c2;

    return 0;
}
```

Figure 26 Test program

```
C c1;
C *cp1 = &c1;
>> Incrementing count for c1 to 2
C *cp2 = new C("new C 1");
>> Incrementing count for new C 1 to 1
cp2 = 0;
>> Decrementing count for new C 1 to 0
>> Deleting new C 1 with 0 references
cp1 = cp2;
>> Decrementing count for c1 to 1
C *cp3 = new C("new C 2");
>> Incrementing count for new C 2 to 1
C c2;
cp1 = &c2;
>> Incrementing count for c2 to 2
>> Deleting c2 with 2 references
>> Decrementing count for new C 2 to 0
>> Deleting new C 2 with 0 references
>> Decrementing count for c2 to -1
>> Deleting c1 with 1 references
```

Figure 27 Test Program Output

```
transformed tree for: int main()
{
  __ef __fsm_tab = { 0xBEEFDEAD, 4, {
      { <offset of c1 + 0>, &C::__dftdt, 1, 16, 0, 0 },
      { <offset of @1 + 0>, &operator delete, -3, 16, 0, 1 },
      { <offset of @2 + 0>, &operator delete, -3, 16, 0, 2 },
      { <offset of c2 + 0>, &C::__dftdt, 1, 16, 0, 3 } } };
  __est __es = { 0, 0, &__fsm_tab, (long int *) 0, 0 };
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c1;"), endl);
  C c1; *C::C(&c1, "c1") , __es.__s = 1;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C *cp1 = &c1;"), endl);
  C *cp1; cp1 = &c1;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp2 = new C(\"new C 1\");"), endl);
  C *cp2; cp2 = (( @1 = ::operator new(16) ?
      __es.__s = 2 , C::C(@1, "new C 1") , __es.__s = 1 : 0 ) , @1);
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp2 = 0;"), endl);
  cp2 = 0;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = cp2;"), endl);
  cp1 = cp2;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp3 = new C(\"new C 2\");"), endl);
  C *cp3; cp3 = (( @2 = ::operator new(16) ?
      __es.__s = 3 , C::C(@2, "new C 2") , __es.__s = 2 : 0 ) , @2);
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c2;"), endl);
  C c2; *C::C(&c2, "c2") , __es.__s = 4;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = &c2;"), endl);
  cp1 = &c2;
  return @3 = 0 , (__es.__s = 3 , C::~C(&c2, 2, 0) , (__es.__s = 0 , C::~C(&c1, 2, 0))) , @3;
}
```

Figure 28 Transformed Expressions Without Smart Pointer Extension

```
transformed tree for: int main()
{
  __ef __fsm_tab = { 0xBEEFDEAD, 7, {
      { <offset of c1 + 0>, &C::__dftdt, 1, 16, 0, 0 },
      { <offset of cp1 + 0>, &ReferenceCounter::dtor, 1, 4, 0, 1 },
      { <offset of cp2 + 0>, &ReferenceCounter::dtor, 1, 4, 0, 2 },
      { <offset of @0 + 0>, &operator delete, -3, 16, 0, 3 },
      { <offset of cp3 + 0>, &ReferenceCounter::dtor, 1, 4, 0, 4 },
      { <offset of @1 + 0>, &operator delete, -3, 16, 0, 5 },
      { <offset of c2 + 0>, &C::__dftdt, 1, 16, 0, 6 } } };
  __est __es = { 0, 0, &__fsm_tab, (long int *) 0, 0 };
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c1;"), endl);
  C c1; *C::C(&c1, "c1") , __es.__s = 1 , c1.rc = 1;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C *cp1 = &c1;"), endl);
  C *cp1; cp1 = (cp1 = &c1, (ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp1)),
      (cp1 , __es.__s = 2 , cp1)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp2 = new C(\"new C 1\");"), endl);
  C *cp2; cp2 = (cp2 = (( @0 = ::operator new(16) ? __es.__s = 4 , C::C(@0, "new C 1") ,
__es.__s = 3 : 0 ) , @0) , (ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp2)),
      (cp2 , __es.__s = 3 , cp2)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp2 = 0;"), endl);
  ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp2)) ,
      (cp2 = 0 , ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp2)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = cp2;"), endl);
  ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp1)) ,
      (cp1 = cp2 , ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp1)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp3 = new C(\"new C 2\");"), endl);
  C *cp3; cp3 = (cp3 = (( @1 = ::operator new(16) ? __es.__s = 6 , C::C(@1, "new C 2") ,
      __es.__s = 5 : 0 ) , @1) ,
      (ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp3)) ,
      (cp3 , __es.__s = 5 , cp3)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c2;"), endl);
  C c2; *C::C(&c2, "c2") , __es.__s = 7 , c2.rc = 1;
  foo();
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = &c2;"), endl);
  ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp1)) ,
      (cp1 = &c2 , ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp1)));
  return @2 = 0 , (__es.__s = 6 , C::~C(&c2, 2, 0) , (__es.__s = 5 ,
      ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp3))) , (__es.__s = 3 ,
      ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp2))) , (__es.__s = 2 ,
       ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp1))) , (__es.__s = 0 ,
      C::~C(&c1, 2, 0))) , @2;
}
```

Figure 29 Transformed Expressions With Extension

Without the transformer extension, the transformed tree for the main function would be as shown in Figure 28. The first line of the transformed function contains a finite state machine table used for exception handling. Each of the 4 entries in the table specifies the action to take should an exception occur during execution of the function. There is an entry for the two local variables requiring destruction, along with the dynamically-allocated storage, in the order they occur in the function. Most of the remaining transformed function is fairly self-explanatory. The state variable `__es.__es` is updated to indicate the progress made, (in other words the state of the function), should an exception occur. The final line handles local destructors, updating the state as each destructor is called in reverse order of declaration. Note that there are no state table entries, state changes, or final destruction code, corresponding to the built-in pointers, as there is no cleanup necessary or possible for them.

Figure 29 shows the transformed tree with the reference counter transformer extension. The first difference is that the state table contains entries for each of the three built-in pointers, calling the `ReferenceCounter::dtor` method. In addition, state changes have been added throughout the function for the built-in pointer declarations. Each declaration or pointer assignment now includes the additional smart pointer functionality that was added as part of the transformation extension. And finally, at the end of the function, the built-in pointers are decremented as they go out of scope, in reverse order of

declaration.

7. Further Work

There are several areas that were not covered by this work, most notably temporaries. The problem with temporaries is that the API currently does not support them very well. There is no model for detecting when a temporary goes out of use, which is necessary in order to correctly apply reference counting. The current model does not include temporaries in the reference count, which is sufficient for some, but not all, cases. Consider an expression such as `cp2 = cp1++;` The initial value of `cp2` must be saved in a temporary prior to the increment of `cp2` in order to be assigned to `cp1`, so the expression would be transformed as follows:

```
cp2 = (@0 = cp1 , cp1 = cp1 + 1 , @0);
```

Applying the smart pointer transformation without taking into account the temporary would yield an incorrect result if the decrement against `cp1` caused the underlying storage to be deleted, resulting in a dangling reference being assigned to `cp2`. The API needs a mechanism for allowing a transformation to determine when a temporary goes out of use so that, for this example, the appropriate reference counting can take place. When such support for temporaries is avaiable, the smart pointer implementation must also be updated to generate temporaries for the modified expressions to avoid multiple evaluation of expressions containing side-effects, as discussed earlier.

Further work also needs to be done in the area of non-implicit scope termination, (for example through a return statement), and exception handling. The current transformation implementation handles reference decrementing only for implicit scope termination, through the state table additions. However, the API does support the capability to handle explicit scope termination, by detecting return statements and using the state information to determine what needs to be done. With respect to exception handling, the current extension implementation does work for simple examples, but not for all cases. Due to time constraints, we did not pursue these areas, but anticipate that the implementation would be fairly straightforward.

8. Conclusion

C++ smart pointers, while similar to built-in pointers, cannot be used interchangeably. Most notably, implicit compiler conversions are not supported for smart pointers. We have proposed that the "smarts" of smart pointers be added to built-in pointers, and presented the expression transformations that would be necessary to implement a reference-counting built-in pointer. Using the Montana API, we have demonstrated a working example of these ideas.

The Montana API interface has proven to be quite complete for the purposes of adding transformation extensions. Other work in this area [Car97] supports this conclusion. We found the API interface and design to be reasonably straightforward and understandable, particularly given the complexity of the problem we were attempting to solve, that of modifying compiler-generated expressions. Nonetheless, adding a transformation extension is not a trivial undertaking, and is more likely to be expected of a class library vendor rather than a casual programmer.

While there is some additional work necessary to allow full support for a reference counting smart pointer implementation, it is clear that the interface is quite capable of handling such language-level extensions. The API definitely need better support for temporaries, both those generated by the compiler and by extensions such as the one we have demonstrated. However, given the flexibility of the interface, this does not seem like a difficult design issue.

9. References

[Alg95] Alger, Jeff; *Secrets of the C++ Masters*, London, England: Academic Press, 1995.

[Bar94] Barton, John; Charles, Phillipe; Chee, Yi-Min; Karasick, Michael; Lieber, Derek; and Nackman, Lee; "CodeStore: Infrastructure for C++-Knowlegeable Tools", Unpublished position paper submitted to OOPSLA 1994 Workshop on Object-Oriented Compilation. http://www.research.ibm.com/softwaretechnology/papers/oopsla94_codestore/position.html

[Car97] Carmichael, Ian; Unpublished working example of using a Montana transformation extension and global program knowledge to de-virtualize virtual function calls.

[Chu94] Churchill, Steve; "Exception Recovery with Smart Pointers", *C++ Report*, January 1994.

[Coh96] Cohen, Shimon; "Lightweight Persistence in C++", *C++ Report*, May 1996.

[Det92] Detlefs, David; "Garbage Collections and Run-time Typing as a C++ Library", *Usenix C++ Technical Conference Proceedings*, June 1992.

[Ede92a] Edelson, Daniel R.; "A Mark-and-Sweep Collector for C++", *Conference Record of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1992.

[Ede92b] Edelson, Daniel R.; "Smart Pointers: They're Smart, but They're Not Pointers", *USENIX C++ Conference Proceedings*, 1992.

[Ede92c] Edelson, Daniel R.; "Precompiling C++ for Garbage Collection", *International Workshop on Memory Management Proceedings*, September 1992.

[GC96] "Great Circle Technology Overview"; Geodesic Systems, 1996.

[Ham96] Hamilton, Jennifer; *Programming with DirectoToSOM C++*, New York, NY: John Wiley, 1996.

[Har96] Harvey, David; "Smart Pointer Templates in C++", 1996, http://web.ftech.net/~honeyg/articles/smartp.htm

[Lip96] Lippman, Stanley; *Inside the C++ Object Model*, Reading, Mass: Addison-Wesley, 1996.

[Mey96a] Meyers, Scott; *More Effective C++*, Reading, Mass: Addison-Wesley, 1996.

[Mey96b] Meyers, Scott; "Refinements to Smart Pointers", *C++ Report*, Nov-Dec, 1996.

[Mil96] Miller, Justin; "Clean Up: C++ Garbage Collection", *Byte*, January 1996.

[Stro91] Stroustrup, Bjarne; *The C++ Programming Language, 2nd Edition*, Reading, Mass. Addison-Wesley, 1991.

[Stro89] Stroustrup, Bjarne; "The Evolution of C++: 1985 - 1989", *Computing Systems*, Summer 1989.

[Stro94] Stroustrup, Bjarne; *The Design and Evolution of C++*, Reading, Mass: Addison-Wesley, 1994.

10. Special References

The following referenced papers are IBM internal documents that we expect will be made publicly available in some form as part of the Montana product. We provide a brief synopsis here.

[Kar96] Karasick, Michael; "So You Want To Write a Tool?", 1996. *Introduces the structure of CodeStore tools that use the query interface and explains the CodeStore programming conventions and idioms.*

[Nac96] Nackman, Lee; "An Overview of Montana", 1996. *Provides an introduction to Montana, it's history, goals, and architecture.*

[Sor96] Soroker, Danny; "Montana White Paper: Extensions", 1996. *Describes the Montana interface for defining and loading extensions.*

[Stre96] Streeter, Dave; "Building Programs with Montana", 1996. *Describes how to use Montana to compile programs and build the Montana database.*

Toba: Java For Applications
A Way Ahead of Time (WAT) Compiler

Todd A. Proebsting Gregg Townsend Patrick Bridges
John H. Hartman Tim Newsham Scott A. Watterson
The University of Arizona *

Abstract

Toba is a system for generating efficient standalone Java applications. Toba includes a Java-bytecode-to-C compiler, a garbage collector, a threads package, and Java API support. Toba-compiled Java applications execute 1.5–4.2 times faster than interpreted and Just-In-Time compiled applications.

1 Introduction

Java [GYT96] is an object-oriented language designed by Sun Microsystems that supports mobile code, i.e., executable code that runs on a variety of platforms. Although the language is interesting in its own right, Java's popularity stems from its promise of "write once, run anywhere." Mobile code proponents envision a future of location-independent code moving about the Internet and running on any platform.

Java's mobility is achieved by compiling its object classes into a distribution format called a *class file*. A class file contains information about the Java class, including *bytecodes*, an architecturally-neutral representation of the instructions associated with the class's methods. A class file can execute on any computer supporting the Java Virtual Machine (JVM). Java's code mobility, therefore, depends on both architecture-neutral class files and the implicit assumption that the JVM is supported on every client machine.

Most JVM implementations execute bytecodes via interpretation or *Just-In-Time* (JIT) compilation, which compiles the bytecodes into machine code at run time. Thus, Java's mobility comes at a price, exacted by the cost of interpreting or JIT-compiling the bytecodes *every time the program is executed*. These systems incur modest to severe performance penalties compared to more traditional systems that compile source code directly to machine code once. For example, a compiled C program runs 1.5-2.2 times faster than the equivalent JIT-compiled Java program, and 2.6-4.2 times faster than an interpreted Java program.

These performance penalties are especially bothersome in non-mobile applications that are run many times without change. To combat these inherent performance penalties we have developed a Java system that precompiles Java class files into machine code. Our system, Toba,[1] first translates Java class files into C code, then compiles the C into machine code. The resulting object files are linked with the Toba run-time system to create traditional executable files. To distinguish our technique from JIT compilation, we have (somewhat facetiously) coined the phrase *Way-Ahead-of-Time* (WAT) compiler to describe Toba. Toba compiles Java programs into machine code during program development, eliminating the need for interpretation or JIT compilation of bytecodes. Although we forfeit Java's architecture-neutral distribution, Toba-generated executables are 1.5-4.4 times faster than alternative JVM implementations.

Toba has several advantages over interpretation or JIT-compilation. First, because Toba runs way-ahead-of-time, rather than just-in-time, the resulting machine code can be more heavily optimized to yield more efficient executables. Second, because Toba creates a C-equivalent to the Java program, the standard C debugging and profiling tools can operate on Toba-generated executables. Third, because Toba executables include all class files used by the application, there is no possibility of an application suddenly ceasing to execute because of a change in available class files. For these reasons we believe that WAT-compilation is valuable for the development and distribution of efficient Java programs.

Toba consists of many components: a bytecode-to-C translator, a garbage collector, a threads package, a run-time library, and native routines implementing the Java API. Toba is a surprisingly small system: the transla-

*Address: Department of Computer Science, University of Arizona, Tucson, AZ 85721; Email: {todd, gmt, bridges, jhh, newsham, saw}@cs.arizona.edu.

[1] Lake Toba is a prominent feature on Sumatra, the island just west of Java.

tor is only 5000 lines of Java; the garbage collector is a modestly-altered version of the Boehm-Demers-Weiser conservative collector [BW88]; the threads package is built on top of Solaris threads; the run-time library is only 6500 lines of C; and the API routines are simply translations of Sun's API class files. Except for dynamic linking, Toba provides a complete Java execution environment.

2 The Java Virtual Machine

The Java Virtual Machine (JVM) defines a stack-based virtual machine that executes Java class files [LY97]. Each Java class compiles into a separate class file containing information describing the class's inheritance, fields, methods, etc., as well as nearly all of the compile-time type information. The Java bytecodes form the JVM's instruction set, and combine simple arithmetic and control-flow operators with operators specific to the Java language's object model. Powerful object-level instructions include those to access static and instance variables, and those to invoke static, virtual, nonvirtual and interface functions. The JVM also includes an exception mechanism for handling abnormal conditions that arise during execution.

The JVM also provides facilities for managing objects and concurrency. The JVM implements a garbage-collected object allocation model, with facilities for initializing and finalizing objects. Concurrency is provided through a thread abstraction. Threads are pre-emptive and scheduled according to priority. A monitor facility provides mutual exclusion on critical sections as well as thread scheduling through wait/notify primitives. Monitors are recursive, allowing a single thread to acquire the same monitor lock multiple times without deadlocking.

3 Toba's Run-Time Data Structures

Java's rich object model requires run-time data structures to describe each object's type and methods. We developed our data structures with both performance and simplicity in mind. They differ in many respects from those of Sun's implementation of Java. For instance, Sun's implementation requires that all object references go through a *handle*, which represents an extra level of indirection, an added inefficiency, and an extra complication. Toba accesses objects directly. The differences are invisible to Java programmers but important to authors of native methods.

3.1 Naming

Toba attempts to preserve Java names in the C it produces, although this isn't always possible. Java names may draw from thousands of different Unicode characters whereas C names are limited to just 63 ASCII characters. Furthermore, some legal Java names such as `enum` and `setjmp` have special meaning in C. When a Java name cannot be used directly as a C name, Toba discards non-C characters, adds a hash-code suffix, and additionally adds a prefix character if the resulting name begins with a digit or other illegal character.

Java method names always require hash-code suffixes. Toba translates each Java method into a C function, and these functions share a global namespace. Because Java methods may be overloaded among and within classes, a hash-code suffix is added to distinguish the methods. The suffix encodes the class name, the method name, and the method signature.

3.2 Data Layout

Java includes eight primitive types: byte, short, int, long, boolean, char, float, and double. Each translates into a primitive C type. (Note that Java's "char" type represents a 16-bit Unicode value.)

All other Java types are *reference* types that subclass the root class, `java.lang.Object`. All reference types are translated into a C pointer type. Each reference points to an object instance, and all instances of a particular class contain a class-pointer to a common class structure. Java has two different kinds of objects: array objects and ordinary objects. The Toba structure for ordinary objects appears in Figure 1. An ordinary object's class descriptor includes the instance size and a flag that indicates it is not an array. The Toba structure for array objects appears in Figure 2. An array's class descriptor includes the element size and its flag indicates that it represents an array. Array instances contain both a length field and a vector of elements.

Each per-class run-time structure has three parts: general information that is needed for all classes (e.g., superclass information), a method table that contains pointers to virtual functions, and a table of class variables. Figure 3 summarizes run-time class-level information common to all classes.

The method table is simply a vector of function pointers and unique method identifiers. The method identifiers are used when invoking interface functions, which must be found at run-time. The structure of the method table is typical of statically-bound object-oriented languages like Oberon-2 [MW91] and C++ [Str86]. Method tables include inherited methods as well as functions defined by the class itself.

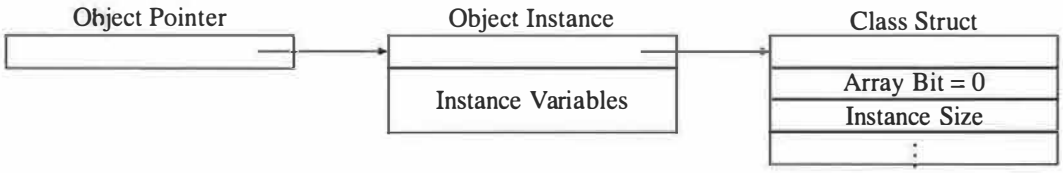


Figure 1: Ordinary Object Structure

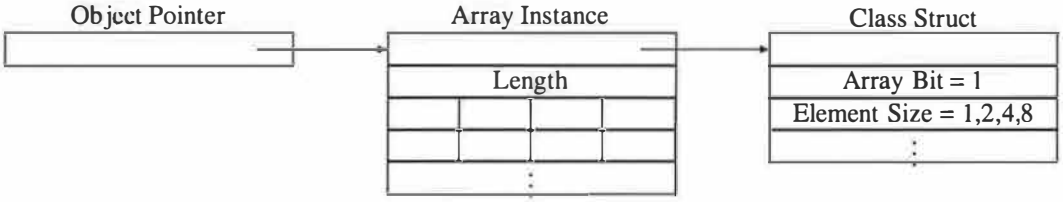


Figure 2: Array Object Structure

initialization flag	Determines if the class has been initialized
other flags	Miscellaneous flags including the Array Bit
class name	Pointer to instance of class `java.lang.String`
class instance	Instance of class `java.lang.Class`
superclasses	Pointer to vector of superclasses for checking subclass relationship
interfaces	Pointer to vector of interfaces
referenced classes	Pointer to vector of referenced classes
array class	Pointer to array class of current class
element class	Pointer to element class, if array class
initializer	Pointer to class initializer function
constructor	Pointer to default instance initializer function
finalizer	Pointer to instance finalizer function

Figure 3: Fields of Class Descriptors

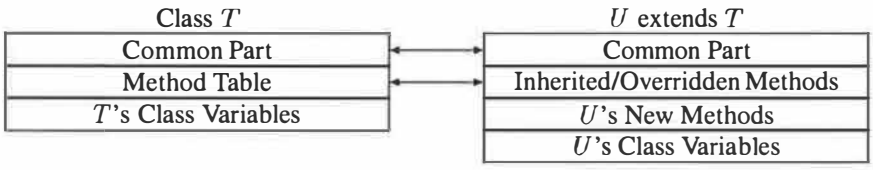


Figure 4: Class/Subclass Structures

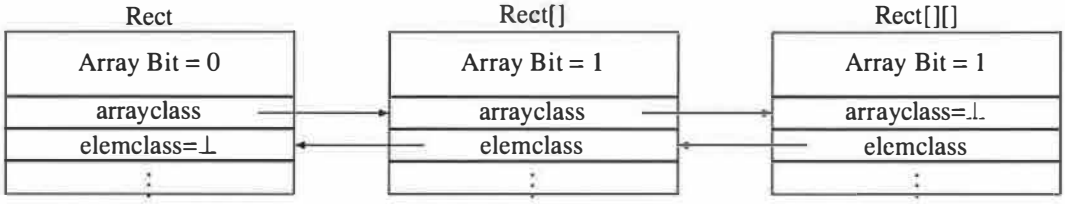


Figure 5: Array Class Descriptors

Class variables exist on a per-class basis, not a per instance basis. Toba-generated programs reference class variables as externals stored in the class structure. Figure 4 shows the class/subclass relationship of class descriptors.

Class descriptors for arrays require special handling. An array of class X ("X[ ]") may be declared by any arbitrary class that imports X. Similarly, an array of arrays of X, X[ ][ ][ ], may be declared by any class that imports X. Descriptors for these array classes must be unique—all instances of X[ ][ ][ ] must share the same class descriptor. Therefore, these array class descriptors must be able to be built at run-time. (It is possible to build them at link-time, but we chose to avoid this complication.) Figure 5 illustrates the simple relationship between the descriptor of a class and the descriptor of an array of that class.

3.3 Referencing Values and Methods

Toba constructs efficient value and method references in C. Assume, for instance, that `r` is an instance of class `rect`. Table 1 summarizes the way Toba references objects and methods in C. Toba-generated C accesses the instance variable `width` as `r->width`. A virtual function call requires an indirection through the method table and requires passing the instance as the first argument. Note that method names include hash suffixes. An interface call utilizes a table-lookup of the appropriate method based on its unique identifier (e.g., 298564082). Static methods and class variables do not require an instance variable. A static method invocation is a simple C function call. Class variables are accessed via the class's run-time descriptor.

4 Code Translation

Toba translates one class file at a time into a C file and a header file. To translate a class file, Toba requires the class files for all of the class's superclasses. To compile a class's resulting C file, header files are necessary from itself, its superclasses, and all imported classes.

4.1 Code Translation

Within class files, methods are encoded in the JVM's byte-coded instruction set. Toba translates each method into a C function. Toba assumes that the class file is valid and verifiable, although it does nothing to confirm this assumption.

The JVM instruction set is stack-based. During execution, (verifiable) bytecode maintains a stack invariant that is critical for translation into efficient C (or native) code: regardless of the previous execution path, at any given point in the program, the stack is always in a consistent state (i.e., the same number and types of values are on the stack). For instance, if along one path to a given program point, P, the stack is empty just prior to executing P, then along *all* paths the stack will be empty just prior to executing P. This invariant means that the depth of the stack and the types of its contents at any point in the program are fixed. A simple traversal of the bytecode can determine this information at compile time. Using this information, the Toba translator is able to turn all stack accesses into references to simple local variables—one per stack location. This eliminates the need for an explicit stack or stack pointer.

Most Java constructs translate simply into bytecode for this stack machine. For instance, the middle column of Figure 6 gives the bytecode for `a=b+c;` assuming that `a`, `b`, and `c` are the first, second and third local variables of the enclosing method. The `iload` and `istore` instructions refer to loads and stores of local variables. Toba creates a C local variable for each JVM local variable.

Figure 6 gives a simple translation of the previous Java statement into C. In the example, `i1` and `i2` refer to the first and second elements of the stack, and `iv1`, `iv2` and `iv3` refer to the first three JVM local variables. Once the stack depths are known, Toba generates naive code. Toba relies on an optimizing C compiler to do copy propagation and register allocation to eliminate useless copies and local variables.

Generating code for each method follows the following outline:

1. Read the bytecode instructions from the class file
2. Compute the stack state at every instruction
3. Note instructions that are exception range entry points and assign labels to them
4. Note jump target instructions and assign labels to them
5. Generate C function header
6. Generate C code for each instruction

Computing stack states requires visiting all instructions. After computing stack state, Toba translates bytecode instructions one at a time.

The Java bytecode supports both direct (conditional and unconditional) branches, as well as indirect jumps. Toba computes all potential targets of direct and indirect jumps, as well as exception handling blocks, in a control-flow analysis. (Verifiable bytecodes are guaranteed to be easy to analyze accurately.) Toba emits a C label before the executable code for each target instruction. To handle indirect jumps and exception han-

Java	Reference Type	Toba-generated Reference
`r.width`	instance variable	`r->width`
`r.flip()`	virtual method	`r->class->M.flip_r_b79qV(r)`
`r.clear()`	interface method	`findinterface(r,298564082)(r)`
`rect.clearAll()`	static method	`clearAll_b7zk4()`
`rect.nrects`	class variable	`cl_rect.V.nrects`

Table 1: Toba-generated References (Omitting C Casts)

Java	Bytecode	Generated C
`a = b + c;`	`iload_2`	`i1 = iv2;`
	`iload_3`	`i2 = iv3;`
	`iadd`	`i1 = i1 + i2;`
	`istore_1`	`iv1 = i1;`

Figure 6: Translating `a = b + c;` into C

dling, a giant `switch` statement wraps each method's generated C code, with each indirect target having its own `case` arm. Thus, indirect jumps translate into C code that sets a program counter variable, jumps to the top of the `switch`, and then dispatches on that variable to the appropriate chunk of code. Unconditional direct jumps become `goto`'s; conditional direct jumps become `if (...) goto L`*n* statements. As an optimization, Toba omits the `switch` wrapper in the absence of exception handling blocks and indirect jumps.

Figures 7 and 8 show a simple Java method along with its translation into bytecode and then into C. The naive code generation algorithm has produced several more assignments than would a human coder, but modern C compilers are good at removing these.

4.2 Exception Handling

The Java Virtual Machine supports exception handling in a manner similar to Ada [Bar84] or C++ [Str86]. Exceptions are *thrown*, either implicitly or explicitly, and are *caught* by the closest matching exception handler. Exceptions that cannot be caught in a procedure require the JVM to unwind the call stack and re-throw the exception in the caller's environment. Re-throwing continues until the exception is caught.

Exception dispatching is based on the execution-time program counter of the JVM. Toba simulates the program counter by assigning values to a local `pc` variable. It is not necessary to set `pc` for every JVM instruction, but only when entering or leaving an exception range (taking into account that jumps can enter the middle of a range).

Toba uses C's `setjmp` and `longjmp` routines to control the call-stack unwinding. For each C function that may catch an exception, Toba creates a small prologue that calls `setjmp` to initialize a per-thread `jmpbuf`. The prologue saves the previous `jmpbuf` value in a local structure; epilogue code restores the old value before the function returns. Toba translates exception throwing into `longjmp` calls that use the `jmpbuf`. Such calls transfer control to the prologue of the nearest function that might handle the exception. This prologue code simply checks a table to determine if, given the type of the exception and the currently active program counter, this procedure can handle the exception. If so, the target label is set to the appropriate handler and execution transfers to the `switch` statement that dispatches indirect jumps. Otherwise, the prologue restores the previous `jmpbuf`, and immediately executes a `longjmp` with this `jmpbuf`.

4.3 Class Initialization

Each Java class may define an initialization routine to be run exactly once. Any of the following events can trigger initialization:

- The first creation of an instance of a class.
- The first invocation of any of a class's static methods.
- The first read or write of any class (not instance) variable.

In the worst case, each of these operations includes checks to determine if the class initializer must be run.

```
class d {
   static int div(int i, int j) {
      i = i / j;
      return i;
   }
}


Method int div(int,int)
   0 iload_0
   1 iload_1
   2 idiv
   3 istore_0
   4 iload_0
   5 ireturn
```

Figure 7: Simple Java Program and Bytecode

```
Int div_ii_3WIeN(Int p1, Int p2)
{                              int div(int, int)
Int i0, i1, i2;                integer stack
Int iv0, iv1;                  integer variables

      iv0 = p1;                init variables from params
      iv1 = p2;

L0:   i1 = iv0;                iload_0
      i2 = iv1;                iload_1
      if (!i2)                 idiv
          throwDivisionByZeroException();
      i1 = i1 / i2;
      iv0 = i1;                istore_0
      i1 = iv0;                iload_0
      return i1;               ireturn
}
```

Figure 8: Sample Toba Output

Calls to allocation routines check a per-class initialization flag. Static methods include checks in their prologue code—no checking is done by the caller. Static-variable accesses include checks of the initialization flag.

Often, these checks are not needed. Toba omits the checks for classes that have no initialization routine.

5 Garbage Collection

Toba's garbage collector is based on the freely-available Boehm-Demers-Weiser (BDW) conservative garbage collector [BW88]. A conservative collector treats every register and word of allocated memory as a potential pointer and traces all memory reached from these pointers. Therefore, the BDW collector does not need type information for the memory it manages. This frees Toba and native routine developers from concerns about memory management.

Our modifications to the BDW collector are relatively minor, affecting about 30 lines of code. First, the BDW collector is a mark-and-sweep collector that requires all threads to be stopped during collection. This proved to be expensive in Toba's thread package (Solaris threads), so we optimized the "stop the world" functionality for the single-threaded case.

Second, the behavior of finalizers and cyclic data structures in the JVM are slightly different from those supported by the BDW collector. The Java language specification (page 231-234 , [GJS96]), allows object finalizers to make previously unreachable objects reachable again, thereby "resurrecting" the objects. Although the BDW collector supported finalization and resurrection of objects, it did not collect cyclic data structures containing finalizable objects. We therefore made another minor modification to the BDW collector to add this functionality.

6 Threads and Synchronization

The JVM defines a priority-based, preemptive thread model that includes synchronization facilities. Toba implements Java threads using Solaris threads, and uses Solaris locks to protect internal critical sections. The biggest problem we encountered when implementing Java threads is that Java allows threads to both suspend each other and to cause other threads to receive an asynchronous exception, such as thread termination. Toba uses UNIX's signal mechanism to handle these asynchronous events, causing the receiving thread to either suspend itself or throw an exception, as appropriate. The problem is that this may cause a thread to block (or even die) in the middle of a critical section, leaving the critical section locked. To eliminate this possibility Toba uses a limited form of roll-forward [MDP96] to allow a thread interrupted by a signal to exit the critical section before handling the signal. Note that this problem also exists with critical sections in the Java code itself; the Java literature does not offer much of a solution other than recommending limited use of these asynchronous thread operations.

Java threads synchronize via monitors. Each object and class has a monitor associated with it, and only one thread at a time may hold the lock associated with a monitor. Condition variables are also provided to allow thread scheduling; the standard wait, notify, and broadcast operations are supported.

An unusual feature of Java monitors is that they are recursive, i.e. the same thread may enter a monitor recur-

sively without deadlock. This implies that Toba cannot implement Java monitors using lock and unlock primitives directly; instead monitors are a more complex data structure containing a lock, a reference count, and the identity of the thread holding the lock. If a thread enters a monitor whose lock it already holds, the reference count is incremented. Similarly, when the monitor is exited the reference count is decremented and the lock only released when zero is reached. If a thread leaves the monitor to wait on a condition, the lock is released and the reference count cleared; when the thread subsequently reenters the monitor the lock is re-acquired, and the reference count is restored.

To reduce synchronization overhead, Toba has an optimized monitor implementation for single-threaded applications. Entering and exiting monitors only affects their reference count; the monitor locks are not used. Should another thread be created, the original thread first locks all monitors that have a positive reference count, thus ensuring mutual exclusion now that there is more than one thread.

7 Performance Results

7.1 Methodology

We tested Toba's performance using using both *application* benchmarks and *micro*-benchmarks. The application benchmarks test the overall system performance, while the micro-benchmarks isolate the performance of individual language features (e.g., exception handling, thread switching, etc.).

We compared Toba's performance to three other systems: Sun's interpreter (JDK version 1.0.2), Sun's JIT compiler system for Solaris, and the Guava JIT compiler (version 1.0 beta 6), by Softway Pty, Ltd. We compared against the Sun interpreter because it is the reference implementation of Java, and against the Guava JIT compiler and the Sun JIT compiler because they are the only other compilation systems for SPARCs of which we are aware. We ran benchmarks on a Sun SPARCStation-20 with 128 MB of memory and two Model 61 SuperSPARC processors. C code was compiled using Sun's commercial C compiler with full optimization (-xO4 -xcg92).

The Guava JIT compiler, the Sun JIT compiler, and Sun interpreter must all do more work at run time than Toba to execute benchmarks. Both systems must dynamically load each class file, and the JIT compilers must compile each method before it can be run. The micro-benchmark times do not include the time to load class files, while application benchmarks do include this time.

7.2 Application Benchmarks

Table 2 describes the application benchmarks. Figure 9 shows the execution times of the benchmarks on the three systems, normalized to the Toba time. Each data point represents the average of ten runs of the benchmark. The JIT system results include the time to compile the benchmark. The Toba-generated benchmarks are 1.5–4.2 times faster than those same benchmarks running under other systems. Toba-generated code runs 2.6–4.2 times faster than programs running under the JDK interpreter, and 1.5–2.5 times faster than the JIT compilers. This speedup results in a tangible improvement in the time to complete the benchmark; the JavaLex benchmark, for example, improved from 159 seconds on JDK and 80 seconds on Guava to only 45 seconds on Toba. The average execution times of the benchmarks, plus standard deviations, are given in Figure 10.

Toba-generated code is faster than Sun's interpreter because compiling class files removes the overhead of interpretation and of dynamic loading. Toba-generated code is faster than the JIT systems because Toba does not incur code generation costs at run time, and, possibly, because the C compiler optimizes code more aggressively than do the JIT compilers. For stand-alone applications that do not rely on dynamic loading, Toba provides large performance benefits over other systems.

7.3 Micro-benchmarks

Table 3 describes the micro-benchmarks used to isolate the performance differences in the systems. These benchmarks are an expanded version of the UCSD Java Microbenchmarks [GP96].

Table 4 shows results of running the benchmarks on each system. For accurate timing, each micro-benchmark was iterated in a loop until the total execution time was at least 5 seconds. This varied between 100 and 100,000,000 iterations, depending on the benchmark.

The results show that Toba outperforms the other systems on almost all benchmarks. For example, Toba is 12–29 times faster than JDK on the arithmetic and class-access benchmarks; this is directly attributable to JDK's use of an interpreter, as Guava and the Sun JIT are nearly as fast as Toba on these benchmarks.

Toba is also usually 0.9–14 times as fast as the other systems at handling exceptions. This is because Toba does not explicitly unwind the stack when an exception is thrown. Instead, Toba implements exception handling via `goto` or `setjmp/longjmp`, depending on whether the handler is within the same method or not. This makes exception handling in Toba extremely fast.

Synchronization is also fast in Toba, particularly in single-threaded programs because Toba optimizes mon-

Application	Description	Input
JavaLex	Lexical analyzer generator that translates regular expressions into finite-state machines that are subsequently translated into Java	Specification that includes 77 patterns
JavaCUP	LALR(1) parser generator that translates context-free grammars into push-down automata that are subsequently translated into Java	Grammar that includes 24 terminals, 32 nonterminals, and 65 productions
javac	Sun's Java compiler that translates Java source programs into class files (bytecode)	Toba source files consisting of 3891 lines of Java
espresso	Translates Java source programs into class files (bytecode)	Toba source files consisting of 3891 lines of Java
Toba	Bytecode-to-C translator described in this paper	Toba's 18 class files (77,718 bytes)

Table 2: Application Benchmarks

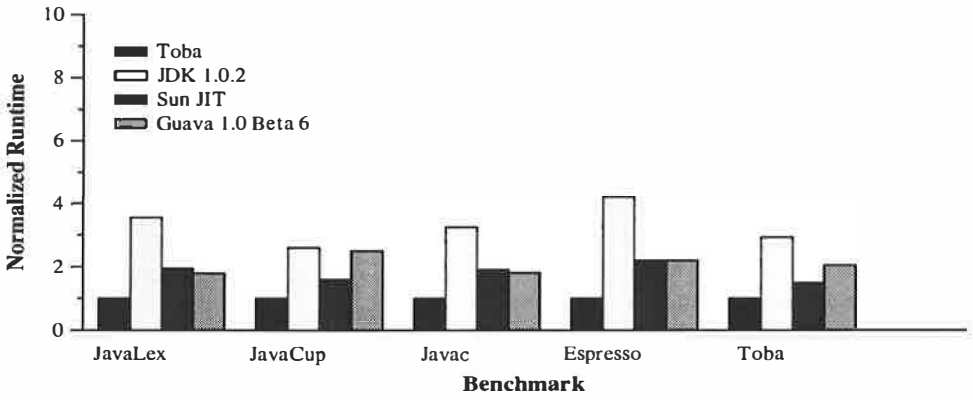


Figure 9: Normalized Application Timings

Benchmark	Toba (sec.)	JDK (sec.)	JDK/Toba	Sun JIT (sec.)	Sun JIT/Toba	Guava (sec.)	Guava/Toba
JavaLex	44.7 ± 0.3	158.9 ± 2.4	3.6	87.5 ± 1.0	2.0	80.0 ± 1.1	1.8
java_cup	2.1 ± 0.05	5.4 ± 0.06	2.6	3.4 ± 0.02	1.6	5.3 ± 0.04	2.5
javac	10.7 ± 0.3	34.8 ± 0.3	3.3	20.4 ± 0.1	1.9	19.4 ± 0.2	1.8
espresso	5.3 ± 0.2	22.3 ± 0.3	4.2	11.9 ± 0.07	2.2	11.7 ± 0.2	2.2
toba	19.3 ± 0.1	56.6 ± 0.5	2.9	28.7 ± 0.1	1.5	39.6 ± 0.4	2.1

Figure 10: Application Benchmark Timings

Arithmetic Benchmarks	
add-int	Add two integers
multiply-int	Multiply two integers
add-double	Add two double-precision floating point numbers
multiply-double	Multiply two double-precision floating point numbers
Class Access Benchmarks	
instance-var	Read an integer instance variable
method-local	Invoke a method defined in the current (`this`) object
method-remote	Invoke a method defined in a different object
method-interface	Invoke an interface method
Exception Handling Benchmarks	
exception-local	Throw and catch an exception within the same method
exception-caller	Throw an exception caught by method's caller
exception-remote	Throw an exception caught by a method ten levels up the call chain
exception-bypass	Throw and catch an exception past an exception handler that does not catch the thrown exception
Synchronization Benchmarks	
sync-block-single	Enter a `synchronized` block in a single-threaded program
sync-method-single	Call a `synchronized` method in a single-threaded program
sync-block-multi	Enter a `synchronized` block in a multi-threaded program
sync-method-multi	Call a `synchronized` method in a multi-threaded program
Miscellaneous Benchmarks	
null-loop	Once around an empty loop
array-assign	Assign to an element of an integer array
thread-yield	Perform yields in 3 separate threads

Table 3: Micro-Benchmarks

itor accesses in this situation. Although single-threaded programs need no synchronization, they may still make use of library classes that use synchronization.

Toba performs slightly worse than Guava on the interface-method invocation benchmark, the integer multiplication benchmark, the instance variable benchmark, and the array assignment benchmark. Toba also performs slightly worse than the Sun JIT compiler on thread yields, since the Sun JIT system does not implement kernel threads or true concurrency. Toba performed better than any of the other systems on all other programs, large and small.

7.4 Code Size

Toba emits naive C code and relies on an optimizing C compiler to do register allocation, copy propagation, and branch elimination to produce efficient code. Table 5 indicates the sizes of the benchmark programs in bytes of class file, lines of C, and bytes of object code. Object code sizes do not include the Toba run-time system, which is a dynamic shared library. This library contains 915,000 bytes of code.

8 Project Status

The Toba system currently runs under Solaris on SPARC workstations. The system includes all of the Java API except for dynamic linking and the graphics and applet libraries. Table 6 summarizes the sizes and implementation languages of its various components.

We intend to port Toba to additional architectures and operating systems. Porting Toba will require thread-specific changes to the run-time system and garbage collector. It will also require OS-specific changes to the run-time system. The bytecode translator and header files will change only minimally.

Toba is the first piece of the larger "Sumatra" project.

Benchmark	Toba (μsec.)	JDK (μsec.)	JDK/ Toba	Guava (μsec.)	Guava/ Toba	Sun JIT (μsec.)	Sun JIT / Toba
add-int	0.10	2.92	29	0.10	1.0	0.18	1.8
multiply-int	0.18	3.14	17	0.17	0.94	0.26	1.4
add-double	0.20	3.72	19	0.46	2.3	0.25	1.3
multiply-double	0.20	3.87	19	0.45	2.3	0.25	1.3
instance-var	0.13	2.43	19	0.12	0.92	0.20	1.5
method-local	0.25	3.11	12	0.36	1.4	0.31	1.2
method-remote	0.30	3.89	13	0.38	1.3	0.36	1.2
method-interface	1.66	3.49	2.10	1.40	0.843	2.26	1.36
exception-local	1.61	4.51	2.80	22.50	14.0	9.30	5.78
exception-caller	6.79	6.52	0.96	30.58	4.50	11.33	1.67
exception-remote	8.52	42.24	4.96	121.37	14.2	26.68	2.66
exception-bypass	8.29	11.73	1.41	32.11	3.87	16.07	1.03
sync-block-single	2.21	14.92	6.75	6.15	2.78	10.73	4.86
sync-method-single	3.17	14.15	4.46	6.49	2.05	10.57	3.33
sync-block-multi	4.37	13.91	3.18	6.14	1.41	10.61	2.43
sync-method-multi	6.23	14.25	2.29	6.44	1.03	10.64	1.71
null-loop	0.03	1.04	30	0.07	2	0.07	2
array-assign	0.21	2.53	12	0.19	0.90	0.24	1.1
thread-yield	75.38	82.31	1.092	79.25	1.05	69.71	0.92

Table 4: Micro-Benchmark Timings

Benchmark	Class-file (bytes)	Emitted C Code (lines)	Object File (bytes)
JavaLex	84,457	25,238	231,816
JavaCUP	119,094	50,297	446,816
javac	508,916	127,678	869,756
espresso	295,281	83,098	674,008
Toba	77,718	23,570	195,836

Table 5: Program Sizes

Component	Implementation Language	Size (Lines)
Bytecode Translator	Java	4723
Run-time Support	C	4130
API Native Routines	C	2809
Toba-specific Garbage Collection	C	30

Table 6: Implementation Details

The Sumatra project is exploring many aspects surrounding the efficient execution of mobile code, with emphasis on efficient implementations of the Java Virtual Machine. We developed Toba to bootstrap our development of the JVM API, threads, and garbage collector, as well as to have fast Java applications.

9 Related Work

Java is a relatively new programming language and virtual machine. We know of no published results describing implementation and performance characteristics. Popular-press reports and commercial advertisements indicate that many development efforts for Just-In-Time (JIT) compilers are underway or have recently completed, but the available information is sketchy.

Compiling higher-level languages to C is not new. Many language systems leverage existing compilers and use C as an intermediate language in the compilation process. Systems for Smalltalk [Git94], SR [And82], Scheme [Bar89], Icon [Wal91], Forth [EM96], SML [TAL90], Pascal [Gil90], Cedar [ADH+89], and Fortran [FGMS90] are well known. For traditionally compiled languages like Pascal and Fortran, translation to C improved portability. For Scheme, Forth, and Icon, translation removed interpretation overhead. Similarly, Toba removes interpretation overhead from Java programs.

Several other projects for compiling Java bytecodes to C are currently underway. `j2c` [And96] is a restricted bytecode to C compiler, currently ported to several platforms. `j2c` (version 1 beta 5) does not support threads, monitors, or network resources. In addition, native routines cannot throw exceptions in `j2c`. Toba does not have these restrictions.

Vortex[DDG+96] is another project that compiles Java bytecodes to C. Vortex provides front ends for C++, Cecil, Modula-3, and Java. These languages are compiled to a common internal representation, and C code is generated from this representation. The Vortex project studies the effectiveness of optimizations for object-oriented languages. The Vortex project reports that Java programs speed up by as much as a factor of 8 as a result of these aggressive optimizations. Toba does not currently perform any of these optimizations. Vortex does not support threads, which has a global impact on performance. No published information is available about other details of Java run-time system support from Vortex.

`Jolt` [Sir96] also compiles Java bytecodes to C. `Jolt` generates a C function for some methods in a class file, and then generates a new class file with these methods marked as native. Method overloading is not supported, and `Jolt` cannot compile class initialization methods. `Jolt` produces class files that are used by the standard Java interpreter. Toba produces stand-alone executables.

10 Availability

The Toba system is freely available via anonymous ftp. All distribution information is described on the World Wide Web at
`http://www.cs.arizona.edu/sumatra/toba/`.

References

[ADH+89] Russ Atkinson, Alan J. Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. Experiences creating a portable Cedar. pages 322–329, 1989.

[And82] G. R. Andrews. The distributed programming language SR—mechanisms, design and implementation. *Software—Practice and Experience*, 12:719–753, 1982.

[And96] Yukio Andoh. j2c. http://www.webcity.co.jp/info/andoh/java/j2c.html, 1996.

[Bar84] J.G.P. Barnes. *Programming in Ada.* Addison-Wesley Publishing Company, 1984. ISBN 0-201-13799-2.

[Bar89] Joel F. Bartlett. Scheme→C a portable Scheme-to-C compiler. Technical Report DEC-WRL-89-1, Digital Equipment Corporation, Western Research Lab, 1989.

[BW88] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, pages 807–820, September 1988.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of OOPSLA'96*, October 1996.

[EM96] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. 1996.

[FGMS90] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 1990.

[Gil90] Dave Gillespie. p2c. p2c is one of several publicly available Pascal to C compilers, 1990.

[Git94] Claus Gittinger. Smalltalk/x. Smalltalk/X is a widely available Smalltalk to C compiler, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996. ISBN 0-201-63451-1.

[GP96] William G. Griswold and Paul S. Phillips. Microbenchmarks for `java`. http://www-cse.uscd.edu/wgg/JavaProf/ javaprof.html, 1996.

[GYT96] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface*, volume 1. Addison-Wesley Publishing Company, 1996. ISBN 0-201-63453-8.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. ISBN 0-201-63452-X.

[MDP96] David Mosberger, Peter Druschel, and Larry L. Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software—Practice and Experience*, 26:1–23, 1996.

[MW91] H. Mossenbock and N. Wirth. The programming language Oberon-2. Technical report, Institute for Computer systems, ETH, 1991.

[Sir96] KB Siram. Jolt. http://substance.blackdown.org/~kbs/jolt.html, 1996.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986. ISBN 0-201-12078-X.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Nov 90.

[Wal91] Kenneth Walker. The implementation of an optimizing compiler for Icon. Technical Report TR 91-16, University of Arizona, August 1991.

A A Larger Example

Figure 11 expands on the example shown earlier by adding exception handling. An implicit branch (from the `try` block to the `return`) has also been added.

Figure 12 gives Toba's translation into C code. Exception handling has enlarged the code significantly, and the effect is especially noticeable because the original example was so small. Besides the boilerplate code that is the same for all exception-catching methods, there are also assignments to `pc` that maintain the JVM program counter and `case` labels used for dispatching a caught exception.

```
class d {
   static int div(int i, int j) {
      try {
         i = i / j;
      } catch (ArithmeticException e) {
         i = j;
      }
      return i;
   }
}
Method int div(int,int)
   0 iload_0
   1 iload_1
   2 idiv
   3 istore_0
   4 goto 10
   7 pop
   8 iload_1
   9 istore_0
  10 iload_0
  11 ireturn
Exception table:
   from   to  target type
     0     4     7   <Class
           java.lang.ArithmeticException>
```

Figure 11: Sample Java Program and Bytecode

```
Int div_ii_3WIeN(Int p1, Int p2)                          int div(int, int)
{
static struct handler htable[] = {                        exception handler list
      &cl_java_lang_ArithmeticException.C, 0, 4, 1,       go to L1 if 0 ≤ pc < 4
};
struct mythread *tdata;                                   thread data pointer
jmp_buf newbuf;                                           jump buffer
void *oldbuf;                                             pointer to previous buffer
volatile int pc;                                          JVM program counter
int tgt;                                                  jump target
Int rv;                                                   return value
Object a0, a1, a2;                                        reference stack
Int i0, i1, i2;                                           integer stack
volatile Int iv0, iv1;                                    integer variables

      iv0 = p1;                                           initialize variables from parameters
      iv1 = p2;

      tdata = mythread();                                 set thread data pointer
      oldbuf = tdata->jmpbuf;                             save old jmpbuf pointer
      tgt = 0;                                            dispatch first to entry point
      if (setjmp(newbuf)) {                               set up jump buffer
          sthread_got_exception();                        exception was caught:
CATCH:    a1 = tdata->exception;                          load exception value
          if ((tgt = findhandler(htable, 1, a1, pc)) < 0)   find handler
              longjmp(oldbuf, 1);                         no handler; pass upward
          }
      tdata->jmpbuf = newbuf;                             register jump buffer for thread

TOP:  switch(tgt) {                                       dispatch entry, ret, or exception

L0:   case 0:
      pc = 0;                                             set pc for exception handling
      i1 = iv0;           iload_0
      i2 = iv1;           iload_1
      if (!i2)            idiv
          throwDivisionByZeroException();
      i1 = i1 / i2;
      iv0 = i1;           istore_0
      pc = 4;                                             reset pc on leaving exception range
      goto L2;            goto 10

L1:   case 1:
      pc = 7;                                             reset pc after catching exception
      i1 = iv1;           iload_1
      iv0 = i1;           istore_0
L2:   case 2:
      i1 = iv0;           iload_0
      rv = i1;            ireturn
      goto RETURN;

}
RETURN:
      tdata->jmpbuf = oldbuf;                             restore previous jump buffer
      return rv;                                          return result
}
```

Figure 12: Sample Toba Output

Making CORBA Objects Persistent: the Object Database Adapter Approach*

Francisco C. R. Reverbel†
reverbel@ime.usp.br
Departamento de Ciência da Computação
Universidade de São Paulo
05508-900 - São Paulo, SP - Brazil

Arthur B. Maccabe
maccabe@cs.unm.edu
Department of Computer Science
University of New Mexico
Albuquerque, NM 87505 - USA

Abstract

This paper discusses a realization of object persistence in a CORBA-based distributed system. In our approach, persistence of CORBA objects is accomplished by the integration of the ORB with an ODBMS. This approach is not limited to pure object-oriented database systems, as the ODBMS may be a combination of a relational DBMS and an object-relational mapper. The design and implementation of an Object Database Adapter that integrates an ORB and an ODBMS with C++ bindings is presented. The ODA uses delegation (rather than inheritance) to connect user-provided implementation classes and IDL-generated classes. Only the user-defined parts of CORBA objects are actually stored in a database. Their IDL-generated parts are dynamically instantiated, in transient memory, by the ODA. Persistent relationships between CORBA objects within a server are not realized at the CORBA level, but at the level of implementation objects. Database traversals and queries can therefore be executed at ODBMS speeds. The paper discusses in some detail a number of implementation issues, such as caching. ODA support to local transactions, ODA interfaces, and CORBA server organization are also examined.

1 Introduction

In spite of its remarkable successes in promoting standards for distributed object systems [14], the Object Management Group (OMG) has not yet settled the issue of object persistence in the Object Request Broker (ORB) environment. The Common Object Request Broker Architecture (CORBA) specification [7] briefly mentions an Object-Oriented Database Adapter that makes objects stored in an object-oriented database accessible through the ORB. This idea is pursued in the Appendix B of the ODMG standard [2], which identifies a number of issues involved in using an Object Database Management System (ODBMS) in a CORBA environment, and proposes an Object Database Adapter (ODA) to realize the integration of the ORB with the ODBMS.

Possibly because this proposal was perceived by many as biased towards object-oriented databases, and hence distant from the mainstream database world, no further OMG specifications have contemplated the ODA approach. Instead, a Persistence Object Service (POS), designed to accommodate the widest possible variety of data stores, was introduced in [8]. So far POS failed to deliver its promise. In response to this fact, the OMG recently issued a request for proposals for POS version 2.0:

> "While the industry posses many products from OMG members that could be considered to be in this space, it is clear that virtually none have compliant POS implementations in their product roadmaps. Most have taken the route of point integrations with ORB products." ([11], page 20)

Meanwhile, recognition that the ODA approach is not exclusive to object-oriented databases seems to have grown in the industry. Object-relational mappers — systems that map C++ classes/objects into relational tables/tuples — have been employed to make relational databases appear as object-oriented ones. Because such mappers implement an ODBMS interface on top of a relational system, they extend to relational databases the applicability of the ODA approach.

The benefits of integrating ORBs and ODBMSs include:

Database Heterogeneity. ORB/ODBMS integration allows the construction of distributed object databases that can be heterogeneous even with respect to the DBMS software running on the database server nodes.

*This research was performed at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 97545, as part of the Sunrise Project.

†Partly supported by a fellowship from the National Research Council of Brazil (CNPq).

"IDL views". Access to database objects through IDL interfaces does not require knowledge of the database schema: changes in the schema are transparent to IDL clients. Interfaces can be defined to expose only data items that certain users are permitted to read or update. IDL interfaces to database objects can therefore play a role analogous to relational views, both for data independence and for authorization purposes.

Language Heterogeneity. Databases can be accessed by CORBA clients written in any language for which a mapping from IDL is defined.

Security. The ORB's remote method invocation mechanism requires much less trust in the client than the data-shipping approach employed by pure object-oriented DBMSs.

This paper discusses the design and implementation of an ODA that integrates an ORB and an ODBMS with C++ bindings. For our purposes, an ODBMS is a system with programming interfaces similar to the ones specified in [2]: it may be a pure object-oriented DBMS (an *OO*DBMS), or a combination of a relational DBMS and an object-relational mapper.

An ODA based on the ideas presented here was developed as part of the Sunrise Project[1] at the Los Alamos National Laboratory (LANL). This adapter has been used by the TeleMed system [4] since mid 1995, and is currently employed by other LANL projects as well. We have implemented it for two ORBs, Orbix and VisiBroker for C++, with ObjectStore as the underlying ODBMS in both cases. Even though these implementations were aimed at a non ODMG-compliant ODBMS, we report our experience in ODMG terms whenever possible.

1.1 The Case for an ODA

ODBMSs integrate database capabilities with an object-oriented programming language. They implement *persistent memory*, a single-level store abstraction of the memory hierarchy. An ODBMS with C++ bindings provides a persistent address space for C++ objects, with heap-style allocation/deallocation. ODBMS programmers manipulate persistent C++ objects in the same way they manipulate objects in the transient heap.

Nevertheless, a CORBA server implemented in C++ cannot simply place in persistent memory the objects it implements. To have the status of a CORBA object, a C++ object must be registered with the ORB, which keeps a per-server-process table of active objects. The details of how C++ objects are registered as CORBA objects are not fully specified by the current release of CORBA.[2] In existing ORBs, CORBA objects are registered upon creation. The following approaches are currently used by ORB implementations:

1. A server may create CORBA objects only via calls to the ORB, usually to the function `BOA::create`.

2. A server can instantiate CORBA objects directly. The constructor of a CORBA object executes IDL-generated code that registers the object with the ORB.

On the other hand, the ODBMS provides its own overloaded form of operator `new`. It requires persistent objects to be created by this operator. If the ORB enforces approach 1 above, then there is clearly no way of placing a CORBA object in persistent memory. If the ORB supports approach 2, one could naively use the overloaded form of operator `new` to instantiate "persistent CORBA objects". This would not work, because the constructor of a persistent object is invoked only when the object is added to the database: "persistent CORBA objects" stored by other processes (including previous runs of the same server program) would not be registered with the ORB. As far as the ORB is concerned, these objects would not be active — no requests would be delivered to them.[3]

To make the ORB and the ODBMS work together, an additional component is necessary. Driven by incoming requests, such a component should activate objects that lie dormant in persistent memory. To allow on-demand activation of dormant objects, it must ensure that object references handed out to CORBA clients contain information on the location of the corresponding objects in persistent memory. Hence this component has to be responsible for the generation and interpretation of references to persistent objects. In the OMG ORB architecture these responsibilities belong to an Object Adapter.

1.2 The Role of the ODA

The primary role of the ODA is to provide CORBA servers with an application-independent way of making CORBA objects persistent. This includes ensuring that references to persistent objects are themselves persistent. In CORBA, *persistence of object references* means that "a client that has an object reference can use it at any time without warning, even if the (object) implementation has been deactivated or the (server) system has been restarted" [7].

[1] See `http://www.acl.lanl.gov/sunrise/sunrise.html`.

[2] The underspecification of a number of server-side issues led to server portability problems [9], which the OMG is about to solve [1].

[3] CORBA distinguishes *object activation* (activation of individual objects within a server) from *implementation activation* (server activation, usually performed by ORB-provided daemons).

With persistence of object references, it makes perfect sense for a client to store an object reference for later use. References to persistent CORBA objects implemented by server X can be stored by server Y (a client of server X), thereby enabling the construction of ORB-connected multidatabases. In such a multidatabase, references to remote objects are used to express relationships between CORBA objects implemented by different servers.

Distributed transactions, in an ORB-connected multidatabase, should be supported by a TP monitor that implements the Object Transaction Service (OTS) specified by the OMG [8]. In the absence of this service, the ODA has the additional role of ensuring that operations on persistent objects are encompassed by local transactions.[4] It interacts with the ODBMS to start and commit (or abort) database transactions.

1.3 Organization of this Paper

The next section motivates and presents the general design of the ODA. Section 3 discusses implementation issues; Section 4 considers transactions; Section 5 examines the ODA interfaces and their typical usage; Section 6 mentions related work; and Section 7 presents concluding remarks.

2 Design Decisions

Our perspective is the one of a third-party implementor, with no access to ORB and ODBMS internal interfaces. Accordingly, our ODA is an add-on to the ORB's native Object Adapter (OA), rather than a replacement for it. Figure 1 shows how it fits into the integrated ORB/ODBMS environment.

Note that the ODBMS is depicted as a separate entity holding persistent objects. This representation exposes the three-tiered nature of the ORB/ODBMS environment: an object implementation — the middle tier — is at the same time a client of the ODBMS and a server to CORBA clients. For simplicity, in a subsequent figure we omit the ODBMS and represent persistent objects within the CORBA server. The reader should keep in mind that "persistence within an object implementation" is a simplified representation of the architecture in Figure 1.

[4]Several *OO*DBMSs, including ObjectStore, do not yet support the resource manager interface required by OTS. This service might also be absent simply because a particular application does not need distributed transactions.

2.1 What to Place in Persistent Memory

A CORBA object has two parts: an IDL skeleton and an user-defined part.[5] The *skeleton* consists of ORB-specific data members and member functions, all of them mechanically generated from an IDL specification. It is an instance of a *skeleton class*, a server-side dispatcher generated by the IDL translator. The user-defined part is the *implementation object*, an instance of an *implementation class* provided by the server writer. The implementation object encompasses the data members and member functions actually defined by the object implementor.[6]

The data members in the user-defined part of a CORBA object are relevant to the application, the ones in the skeleton part are relevant to the ORB only. If we employ an ODBMS to make CORBA objects persistent, we should certainly keep their implementation objects in persistent memory. Should we also place their skeleton parts in persistent memory? An obvious reason for not doing so is waste of database space, specially in the case of fine-grained objects.[7] Stronger reasons are:

ORB independence. Keeping ORB-specific data members in persistent memory ties the database to a particular ORB implementation. As ORB products evolve, these data members may change with ORB releases. Databases with ORB-specific information would then have to go through a schema evolution process.

Performance. Assuming that CORBA objects are reference counted,[8] the skeleton part of a CORBA object holds its reference count, which is updated by the primitives `duplicate` and `release`. Placing reference counts in persistent memory means encompassing these primitives by update transactions. Every operation that receives or returns a reference to a persistent object would then require an update transaction, because parameter passing involves `duplicate` and `release` calls.

Only the user-defined parts of CORBA objects should be placed in persistent memory. As the ODA activates

[5]We are not considering the case of CORBA objects implemented with the Dynamical Skeleton Interface.

[6]Terminology could be better here, as *implementation object* is easily confused with *object implementation*. The former is an instance of an implementation class, the latter is the OMG term for a CORBA server. We prefer the vocabulary adopted by [5] — *servant* for implementation object, *servant class* for implementation class — but refrain from using it, because the new Portable Object Adapter specification [1] has assigned another meaning to the word servant.

[7]Besides ORB-specific data members, the skeleton part of a CORBA object typically has a pair of hidden vbase and vtable pointers for each interface class in the object's inheritance chain up to `CORBA::Object`.

[8]Although CORBA does not specify such implementation details, most (if not all) ORB implementations keep a reference count per object.

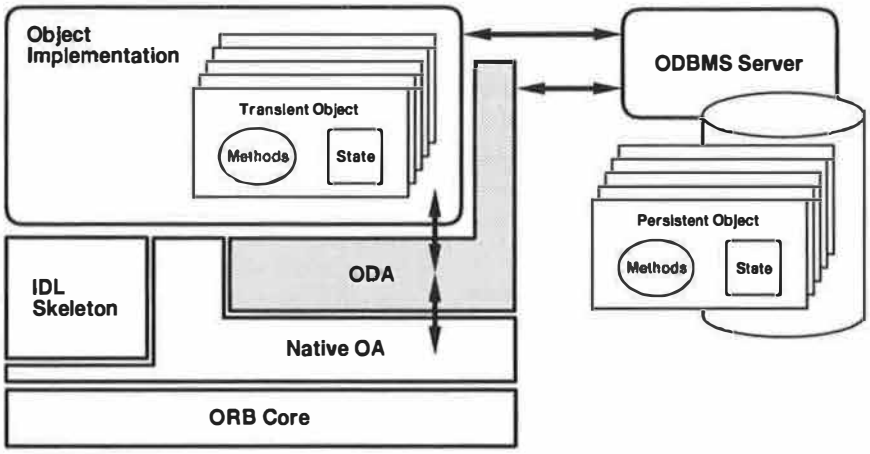


Figure 1: The Object Database Adapter.

and deactivates objects, it should dynamically instantiate and release their skeleton parts, allocated in transient memory. These observations lead us to a clear choice with respect to the relationship between skeletons and implementation objects.

2.2 Delegation, Not Inheritance

Figure 2 shows the alternatives commonly used to connect the parts of a CORBA object. In the *inheritance approach*, the object implementor derives implementation classes from IDL-generated skeleton classes. In the *delegation approach*, also known as *tie approach*, instances of IDL-generated skeleton classes are called *tie objects*, or simply *ties*. Each tie holds a reference to an implementation object to which it delegates operations. While inheritance imposes identical lifetimes to both parts of a CORBA object, delegation allows implementation objects to outlive their skeleton objects. We therefore choose delegation as the interface implementation approach supported by the ODA.

2.3 Pseudopersistence

Our decisions can be summarized as follows:

- The ODA supports persistent CORBA objects implemented with the delegation approach.
- CORBA servers keep only implementation objects in persistent memory.
- The ODA is responsible for dynamically instantiating and releasing transient ties to persistent implementation objects, so that full CORBA objects are available whenever they are needed.

Even though "persistent CORBA objects" are not fully kept in persistent memory, to their clients they appear as long-lived objects. Accordingly, we call this scheme *pseudopersistence*. In what follows, a *pseudopersistent tie*, or simply *p-tie*, is a transient tie to a persistent implementation object.

As any tie, a p-tie has a data member that specifies the implementation object to which the tie delegates operations. In a regular tie, this data member is a C++ pointer or reference. In a p-tie, it must be an ODBMS reference (`d_Ref`), for it points to an implementation object in persistent memory.

When a p-tie is instantiated, one must initialize its `d_Ref` data member. To support the instantiation of a p-tie given a CORBA object reference, the ODA embeds a `d_Ref` to an implementation object into every CORBA reference it generates. This embedding takes advantage of the `id` (also known as `ReferenceData`) field of the object reference. The `id`, an octet sequence opaque to the ORB core, contains identification information local to the server in which the CORBA reference was generated. References to p-ties are generated and interpreted by the ODA, which embeds `d_Ref`s into their `id`s.

Figure 3 illustrates the pseudopersistence scheme. A request to a dormant object arrives through the ORB core (1), causing an upcall to an ODA-provided object activation function. The `id` field of the target object reference is passed as a parameter to this function. This `id` contains a stringfied `d_Ref` to a persistent implementation object. The ODA extracts the `d_Ref` from the `id` and passes it as an argument to an instantiation function (2), which constructs the target CORBA object as a p-tie to the implementation object specified by the `d_Ref`. The incoming request then reaches the target object as an upcall through the IDL skeleton (3). At the end of the operation, another upcall to the ODA (4) causes the target object to be released.

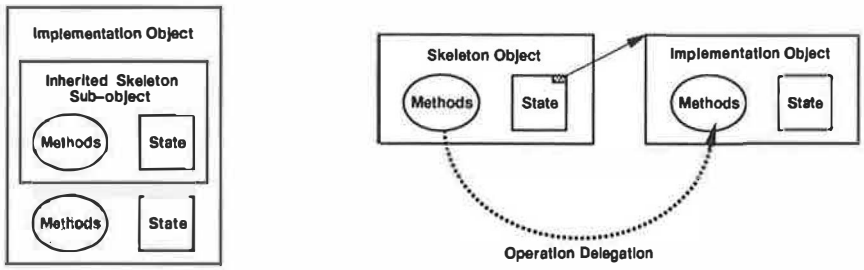


Figure 2: Interface implementation approaches.

In Figure 3, the object activation upcall happens because the target of an incoming request is a dormant object. Upcalls also happen in the case of dormant objects referenced by request parameters, or by strings passed to `string_to_object`.

Pseudopersistence should be understood in the context of the architecture in Figure 1. Persistent implementation objects do not live within a CORBA server, as the simplified representation in Figure 3 may suggest, but in a database server. Multiple CORBA servers (middle-tier processes) may be clients of a database server; they may or may not run in the same node as the database server. Moreover, a persistent implementation object may be shared by multiple p-ties, each in the address space of a different middle-tier process.[9]

3 Implementation Issues

The ODA is implemented as a library that uses and extends the services of the native OA. It requires changes on the IDL translation process, which must become ODA-aware. These changes, as well as the actions of the ODA library, are examined below.

3.1 IDL Translation Issues

- Any tie class has a data member that references the implementation object to which ties delegate operations. This data member is usually a C++ pointer or reference. In the case of a p-tie class, however, it must be a `d_Ref`.

[9]To exemplify: consider a persistent CORBA object implemented by an Orbix server whose activation mode is *per-client-process*, and suppose that multiple clients are concurrently using this object. Every client interacts with its own middle-tier process, a distinct execution of the same server program. Each middle-tier process has a p-tie that incarnates the persistent CORBA object. All these p-ties share the same persistent implementation object, which is managed by the ODBMS.

- Code to support the management of p-ties by the ODA library must be generated within every p-tie class. In our implementation, p-tie constructors and destructors perform ODA-related actions. Moreover, each p-tie class makes available to the ODA library a static function for p-tie instantiation.

The constructor of a p-tie class embeds into the p-tie's `id` a stringfied `d_Ref` to the p-tie's implementation object. It also registers the p-tie with the ODA library; the p-tie will be eventually unregistered by its destructor. The p-tie instantiation function receives a `d_Ref` to a persistent implementation object and creates a new p-tie to this object.

Special translation requirements do not necessarily mean another IDL translator. Our ODA implementation actually employs the IDL translator provided by the ORB, complementing it with macros. The object implementor annotates the server code with ODA-defined directives, which macro-expand into p-tie class definitions. No changes are made to any files generated by the IDL translator: ODA directives are placed only in user-written files, and typically within server headers. In what follows, an *ODA-generated function* (*ODA-generated class*) is a function (class) defined by the macro expansion of an ODA directive.

3.2 ODA Actions

- The ODA library receives object activation upcalls from the native OA, forwarding each such upcall to the appropriate p-tie instantiation function.

- At the end of every operation, after any results were marshaled into a reply message, the ODA library issues `release` calls on all p-ties instantiated while the current request was being serviced.

Because the number of implementation objects in a database is potentially very large, a CORBA server can-

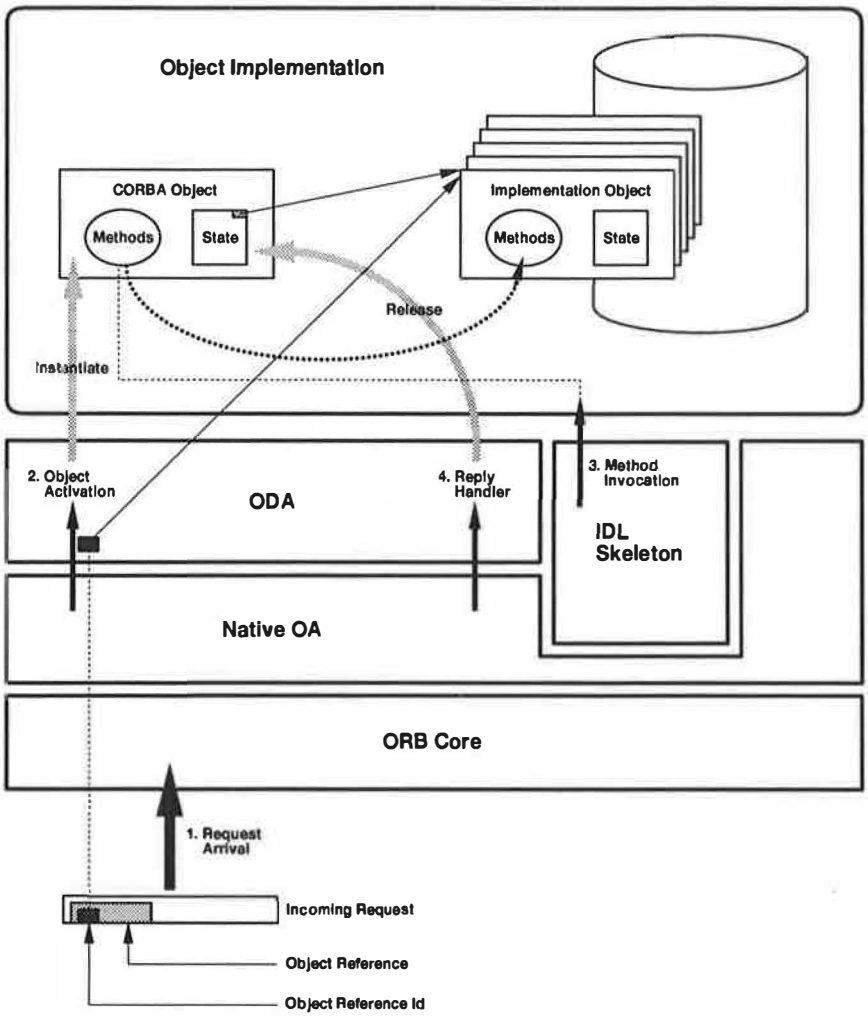


Figure 3: The pseudopersistence scheme.

not keep in-memory ties to all the persistent implementation objects it touches during its execution. The last item above addresses the need of releasing p-ties from time to time. Each p-tie is instantiated with a "net reference count" of zero — an initial reference count of one, plus a pending `release` call, to be performed by the ODA at the end of the operation. Unless the server code issues `duplicate` calls on them, p-ties have short lifetime: they exist while a request is being serviced. Whenever a discarded p-tie is needed again, an equivalent to it will be instantiated by an object activation upcall.

3.3 Caching P-ties

Releasing *all* p-ties at the end of every operation appears unreasonable, since the ODA only needs to ensure that these p-ties will be eventually released. Postponing their destruction would avoid the costs of successive p-tie re-instantiations. Our ODA implementation actually caches the last N p-ties it instantiated, where N is a configurable parameter. At the end of every operation the ODA brings the number of p-ties down to $N + \delta$, keeping the most recent ones. (The δ accounts for any `duplicate` calls that might have been issued by the server code.)

Caching p-ties makes sense if the ODBMS ensures that their `d_Ref` data members remain valid across transactions. So far we have ignored database transactions, this topic will be discussed in Section 4. Let us assume, by now, that transactions are started and committed (or aborted) by means external to the CORBA server, and that each operation is encompassed by an individual transaction.

Does a `d_Ref` from transient to persistent memory retain its validity across transactions? The ODMG standard leaves the answer to the discretion of the ODBMS implementor. In most ODBMSs, such a reference cannot be used in between transactions, but does remain valid across transactions. This being the case, the ODA should

cache p-ties.

With caching, the CORBA server must have a way of forcing the removal of objects from the cache. Accordingly, the ODA provides a function that receives a `CORBA::Object_ptr` and immediately deletes the corresponding p-tie. This function, `ODA::Delete`, is intended to be called by destructors of persistent implementation objects, with the purpose of avoiding dangling p-ties.

3.4 Converting Implementation Objects to CORBA Objects

The ODA must provide the server code with the means for obtaining a CORBA object given its implementation object. For each association

(interface, implementation_class)

there is an ODA-generated function that takes a `d_Ref` to an implementation object and returns a reference (of type *interface*`_ptr`) to the corresponding CORBA object. To avoid multiple p-ties to an implementation object, this function is not implemented as a mere call to p-tie instantiate. It first checks it if a p-tie to the implementation object already exists in the server's address space, then it returns a `duplicated` reference to either an existing p-tie or a newly instantiated one.

A non-standard `bind` function, present in various ORBs, could be used to perform the check mentioned above. Given a `d_Ref` to an implementation object, one would convert it to string and obtain an `id`, which would then be passed as an argument to `bind`. The ODA does not use this approach. Instead, it keeps pairs

(`d_Ref`*, p-tie_address)*

in its own table of active p-ties, which it hashes by `d_Refs` with a hash function provided by the ODBMS.

3.5 Non-standard ORB and ODBMS Features Employed

The ODA relies on the delegation approach, which is mentioned — but not mandated — by the current revision (2.0) of CORBA. Orbix and VisiBroker are examples of commercially available ORBs that support delegation. Both admit direct instantiation of ties, automatically registering newly instantiated ties as active CORBA objects. The new Portable Object Adapter (POA) specification [1] fully standardizes the delegation approach. It requires compliant ORB implementations to support both the inheritance and the delegation approach.

Because CORBA 2.0 describes object activation in very general terms, existing ORB implementations vary widely on their support to object activation. The ODA builds upon the native OA's object activation capabilities. Its Orbix implementation uses a `LoaderClass` instance; the VisiBroker implementation uses an `Activator`. The new POA specification completely defines object activation. With the POA, future ODA implementations can employ a standard facility, the `InstanceActivator` interface.

Various ORBs provide non-standard "event handling" or "request/reply intercepting" mechanisms. The ODA needs such a facility both to release p-ties and to manage database transactions in the absence of OTS (see Section 4). Its Orbix implementation uses a `Filter`; the VisiBroker implementation uses an `EventHandler`. The OMG has recently introduced *request level interceptors* [10] as an extension to the ORB core, and is actively working to complete the specification of this facility.

From the ODBMS, the ODA requires a means of converting `d_Refs` to strings and vice-versa. Although supported by many ODBMSs, this feature is not in the ODMG standard. Caching of p-ties requires more: `d_Refs` must remain valid across transactions.

4 Transactions

Any access to persistent memory has to be performed within a transaction. Leaving to implementation objects the responsibility of starting and committing (or aborting) transactions is not an option, because accesses to persistent memory happen both before and after these objects' methods are called:

- In order to delegate an operation to its implementation object, a p-tie must access persistent memory. The p-tie must dereference its `d_Ref` data member, which points to persistent memory.
- Marshaling of operation results into a reply message may involve accesses to persistent memory.

Usage of OTS [8] would ensure that not just the user-provided implementation code, but also request dispatching and parameter marshaling code, would be executed within transactions. Since OTS interacts directly with the local resource manager (the ODBMS), transactions would be started and committed (or aborted) by means external to the CORBA server.

If OTS is absent, the ODA must take the responsibility of starting and committing (or aborting) *local* transactions. Not with the aim of performing distributed two-phase commit, but just to ensure that a transaction will be active whenever an operation is dispatched, and will remain active till the operation results are marshaled into a reply message. We did not have OTS, so this was our scenario.

4.1 Support to Local Transactions

The ODA manages local transactions by employing ORB-specific "event handling" or "request/reply intercepting" facilities. Its default transaction mode is *transaction per operation*: an "incoming request pre-marshal" handler starts a transaction as soon as a request arrives, an "outgoing reply post-marshal" handler ends the transaction just before the reply is sent. An operation implementation may specify if the current transaction will be committed or aborted at the end of the operation. By default, the ODA commits the transaction. Under control of the server code, the ODA may also switch to another transaction mode, which allows multiple operations to be grouped into a single transaction.

Because ObjectStore requires the transaction type (read-only or update) to be specified when a transaction starts, update operations must be registered with the ODA. Registration of update operations is typically done by the server mainline. By default, the ODA starts read-only transactions. In the case of operations previously registered as update operations, it starts update transactions.

5 ODA Interfaces and Usage

The CORBA server interacts with the ODA through a very small API. Besides ODA-generated functions that return an *interface*_ptr given a d_Ref and vice-versa, there are just a few static functions available to the server code:

- `ODA::initialize`
- `ODA::register_update_ops`
- `ODA::Delete`
- `ODA::multi_op_transaction_mode`
- `ODA::abort_transaction`
- `ODA::commit_transaction`

Note that there is no specific function to create or activate a persistent CORBA object: object activation may occur as a side effect of the conversion of a `d_Ref` into CORBA object reference.

Given an interface class `X` and an implementation class `X_i` to which `X` delegates operations, the function

```
X_ptr ODA_X_i_to_X(const d_Ref<X_i>&);
```

translates a `d_Ref<X_i>` into the corresponding `X_ptr`. This function, defined at the file scope, is generated by the ODA directive that "ties together" `X` and `X_i`. A member function of the ODA-generated p-tie class performs the reverse translation (to `d_Ref<X_i>`).

The ODA is not an intrusive presence in the programming environment. In our experience, the vast majority of ODA calls is performed to obtain an *interface*_ptr from a `d_Ref`. Except for these, ODA calls are relatively rare in the server code. `ODA::initialize` is called once, by the server's mainline. Calls to `ODA::register_update_ops` typically appear in the server's mainline only, and would not be necessary in the case of an ODMG-compliant ODBMS. `ODA::Delete` is invoked from destructors of persistent implementation objects. In the default transaction mode, user-provided methods do not normally call transaction management functions.

5.1 Server Organization

Persistent relationships between CORBA objects within a server are actually realized by relationships between their corresponding implementation objects. When traversing database relationships or performing a database query, the server code deals only with persistent implementation objects, not with full CORBA objects. Such a traversal or query is therefore executed at ODBMS speeds. Consider, for example, the case of an operation that performs a search for a particular object within a collection of objects. The whole search is performed at the ODBMS level, without CORBA-activating any of the objects of the collection. Its result, a `d_Ref` to particular implementation object, is then converted to CORBA object reference and passed back to the client. When the server code calls the ODA to perform such a conversion, it obtains a `duplicated` reference to a CORBA object managed by the ODA. Whether this object was just activated or was already in the ODA cache is irrelevant to the server code, which in either case assumes the responsibility of releasing the reference.

Persistent relationships between CORBA objects in different servers are realized via stringfied CORBA references stored in persistent memory. These references must be explicitly converted back to its native form for usage. Note that any database containing CORBA object references is ORB-dependent, because these references are ORB-dependent. ORB independence is lost when we move on to an ORB-connected multidatabase.

5.2 Inheritance Issues

Consider the following IDL interfaces:

```
interface X {  ...  };
```

```
interface X1 : X {
    ...
};

interface X2 : X {
    ...
};

interface Y {
    readonly attribute X x;
    ...
};
```

Interface X defines operations that are common to both X1 and X2. Attribute x of Y has interface type X, but its most derived interface is either X1 or X2.

A natural organization for the corresponding persistence-capable implementation classes[10] would be:

```
class X_i : public d_Object {
    // abstract class
    ...
};

class X1_i : public X_i {
    ...
};

class X2_i : public X_i {
    ...
};

class Y_i : public d_Object {
public:
    X_ptr x();
    ...
private:
    d_Ref<X_i> x_i;
    ...
};
```

X_i is an abstract class: any instance of this class is an instance of either X1_i or X2_i. Class Y_i holds an ODBMS reference to an instance of X_i in its private data member x_i. The attribute accessor Y_i::x() returns a CORBA reference to the object whose implementation is x_i.

Note, however, that there is no ODA-generated function that takes a d_Ref<X_i> and returns an X_ptr. The ODA provides this conversion function only when the interface skeleton and the implementation class are tied together by delegation. This is never the case for an inherited implementation class, such as X_i. In the example above, there are ODA-generated conversion functions from d_Ref<X1_i> to X1_ptr and from d_Ref<X2_i> to X2_ptr.

ODA users solve this problem by defining a virtual member function, say get_X_ptr(), in class X_i. This function, declared as pure virtual in X_i, is redefined by the derived classes X1_i and X2_i as below:

```
X_ptr X1_i::get_X_ptr() {
    return ODA_X1_i_to_X1(d_Ref<X1_i>(this));
}

X_ptr X2_i::get_X_ptr() {
    return ODA_X2_i_to_X2(d_Ref<X2_i>(this));
}
```

If the inheritance chain were longer, all abstract implementation classes would define get_X_ptr() as pure virtual.

6 Related Work

Work recently concluded at the OMG, in the context of the ORB Portability Enhancement RFP [9], has resulted in a Portable Object Adapter [1] that will reduce the ODA dependencies on non-standard ORB features. Earlier ORB portability proposals [5, 3] included a Server Framework Adapter (SFA) and an ODMG model for SFA. Our pseudopersistence scheme is essentially a realization of the ODMG model for SFA, as outlined in the Appendix C of [3].

A number of ORB and ODBMS vendors has announced plans for the integration of their products; some of these integrated solutions are already being delivered. Probably the first one was Iona Technologies's Orbix+ObjectStore Adapter (OOSA) [6], whose beta release became available by late 1995. Since then, Iona has integrated Orbix with Versant, and has announced plans for integrating Orbix with O2 and with Persistence.

Iona's OOSA takes advantage of the particular way CORBA objects are laid out by the ORB. In Orbix, not all data encapsulated by a CORBA::Object instance appears directly in its data members. Instead, a data member of CORBA::Object points to an auxiliary object. Some of the "logical" data members of CORBA::Object are actually in this auxiliary object. The reference count is one of them.

Unlike the ODA, which stores only implementation objects, OOSA actually stores CORBA objects in ObjectStore databases. A CORBA object, however, is not

[10] We adopt the convention of naming implementation classes by appending an "_i" to the corresponding interface names.

stored in their entirety: to avoid the performance penalty of having reference counts in persistent memory, OOSA does not store the auxiliary object in the database. Instead, it dynamically instantiates auxiliary objects as persistent CORBA objects are made available in ObjectStore's client cache. When such an auxiliary object is instantiated, the corresponding CORBA object is inserted into the per-process table of active objects maintained by Orbix. This approach allows persistent CORBA objects to be implemented either by inheritance or by delegation. It also allows object relationships to be expressed in terms of CORBA objects, not just in terms of implementation objects. Its disadvantages are some waste of database space, ORB-dependent databases, and the performance penalty of object activations triggered by database accesses.

7 Concluding Remarks

We have presented the design and implementation of an ODA that allows execution of database traversals and queries at the full speed of the underlying ODBMS. Only what needs to be persistent is kept in persistent memory; ODA users are not forced to store ORB-specific information persistently. Databases are ORB-independent unless the user explicitly places ORB-specific data (such as stringfied object references) in persistent memory. Finally, the ODA design appears to be general enough to be applicable to any ODBMS. ObjectStore's virtual memory-based architecture makes it different from all other ODBMSs in many aspects. That the ODA design can be described in ODMG terms, and yet be implemented for ObjectStore, is strong evidence of its applicability to any ODBMS.

The ODA's pseudopersistence scheme appears to be an optimal solution for integrated ORB/ODBMS environments in which object relationships are mostly confined within a CORBA server. In such a scenario, there is no reason to express database relationships at the CORBA level, as they are much more efficiently realized at the level of implementation objects.

The motivation for representing database relationships at the CORBA level might arise in the context of an ORB-connected multidatabase with many cross-server references. Expressing persistent relationships between objects in different servers via stringfied CORBA references placed in persistent memory may be inconvenient in this case. Consider, for example, a situation in which it would be desirable for a server to have a persistent and homogeneous collection of object references, whose elements may refer to either local or remote objects. This is not possible in the pseudopersistence scheme. Instead of a uniform collection, two distinct sub-collections must be used: one with `d_Ref`s to local implementation objects, other with stringfied CORBA references to remote objects. Intra-server references and inter-server references could be unified if the Object Adapter provided support for persistently representing *both* at the CORBA level. To be useful, this unification should allow transparent use of stored CORBA references to invoke methods on possibly remote objects. Note, however, that incurring the cost of such a unification — the performance penalty of expressing intra-server references at the CORBA level — makes sense only if cross-server references occur much more than intra-server references.

References

[1] BEA, DEC, Expersoft, HP, IBM, ICL, IONA, Novell, SunSoft, and Telefónica I+D. *ORB Portability Joint Submission, Draft 14.* OMG Document orbos/97-04-04, April 1997.

[2] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann, 1996.

[3] DEC, Expersoft, HP, IBM, ICL, IONA, Novell, SunSoft, and Telefónica I+D. *ORB Portability Joint Submission, Draft 5.* OMG Document orbos/96-12-02, December 1996.

[4] D. W. Forslund, R. L. Phillips, D. G. Kilman, and J. L. Cook. Experiences with a distributed virtual patient record system. *Journal of the American Medical Informatics Association, Symposium Supplement*, 1996.

[5] HP, IBM, Novell, and SunSoft. *Server Framework Specification (ORB Portability Submission).* OMG Document orbos/96-05-03, May 1996.

[6] Iona Technologies. *Object+ObjectStore Adapter — Beta Release Documentation.* Dublin, Ireland, 1995.

[7] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Revision 2.0, July 1995.

[8] Object Management Group. *CORBAservices: Common Object Services Specification.* Revised Edition, March 1995. Updated November 1996.

[9] Object Management Group. ORB Portability Enhancement RFP. OMG Document 95-06-26, June 1995.

[10] Object Management Group. *CORBASecurity*. Version 1.1, OMG Document Numbers 96-08-03 through 96-08-06, July 1996.

[11] Object Management Group. Persistent Object Service, version 2.0 — Request For Proposal (Draft). OMG Document orbos/97-04-07, December 1997.

[12] F. C. R. Reverbel. *Object Database Adapter Programmer's Guide and Reference Manual.* Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, NM, August 1996.

[13] F. C. R. Reverbel. *Persistence in Distributed Object Systems: ORB/ODBMS Integration.* PhD thesis, University of New Mexico, Computer Science Department, Albuquerque, NM, May 1996.

[14] S. Vinosky. CORBA: Integrating diverse applications within heterogeneous environments. *IEEE Communications*, 14(2), February 1997.

Obtuse, a scripting language for migratory applications

Robert P. Cook
Dept. of Computer and Information Science
University of Mississippi
www.cs.olemiss.edu/~bobcook; bobcook@cs.olemiss.edu

Abstract

This paper discusses the design and implementation of Obtuse, a scripting language for migratory applications. The paper reviews the pertinent ActiveX technology that provides the runtime object infrastructure. Then we discuss the Obtuse object model and present an overview of the language. Next, several sample programs are used to illustrate the concepts. Finally, we review some of the problems with DCOM, based on our experience.

Keywords: scripting language, Obtuse, migratory applications, Obliq, distributed systems

1. Introduction

Obtuse was designed by the author, inspired by Cardelli's Obliq [1] and Bharat's Visual Obliq[2] systems, and implemented as part of a summer research appointment at Microsoft Corporation. The goal was to explore the potential of several core ActiveX technologies [3,4,5], including COM (Component Object Model), Automation, and DCOM (Distributed Component Object Model).

Obtuse is unique in two respects; first, in its synergistic use of ActiveX technology and second, in its ability to transfer the state of a Visual Basic form from one machine to another. A **migratory application** is one that can transfer program state (including the user interface) to different Internet locations under program control. Other terms used in the literature are **mobile** or **transportable agents**. Obtuse is also a **scripting language**; that is, it defines sentences capable of being executed as fine-grained code fragments.

As an example of transferring UI state from one machine to another, consider the Visual Basic (VB) form in Figure 1, which consists of an edit control and a button. The form is used in a simple, routing-slip application. The user can type a command line, such as "obtuse poll m1 m2 m3" to initiate execution. The list (a routing slip) represents a sequence of computers to visit. The DCOM infrastructure is utilized by the Obtuse runtime to implement the remote activation that is necessary to support the routing-slip application.

The form is circulated to the machines in the order listed. The accumulated comments are available to each recipient and the completed form, with all comments, is returned to the source machine. When one user clicks the OK button, the form is moved to the screen of the next computer in the list.

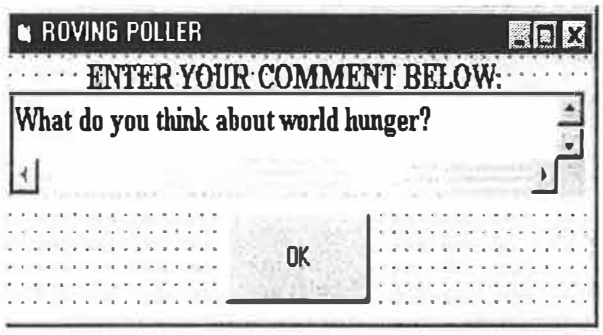


Figure 1. User Interface for a Roving Poller

We refer to programs, such as the routing-slip example, as **in-your-face** applications. When one user clicks the routing slip's OK button, the document appears instantaneously on the screen of the next recipient. Most word processors also support routing slips by using e-mail as the transport mechanism. However, users are only notified if an e-mail client is executing at a site and if they decide to read their mail.

In the routing-slip application, the code, together with its execution state, can also migrate from one machine to another with the form. Obtuse implements program migration by exposing threads and contexts (a program's global variables) as COM objects.

Figure 2 lists a simple Obtuse program that moves itself from one machine to another. In Obtuse, a running program is a collection of COM objects, which support an Automation interface. The sample program creates a thread and a context object on a remote machine. Next, the Fork method of the running-thread object is invoked to clone the program's state. At this point, there are two threads executing, one locally and one remotely.

However, since they have duplicate contexts, any object references in one are duplicated in the other. As a result, the remote thread can access objects on the parent machine in a location-opaque fashion.

```
// Note that variables are "typed" at runtime
var me, thread, context, where;
me := self;              // a reference to the executing thread
print( "parent process starting");
where := "louie.cs.olemiss.edu";
                                  // create a remote thread
thread := object("Bob.Thread", where);
                                  //create a remote context
context := object("Bob.Context", where);
                                //duplicate myself at "where"
if me.Fork{thread, context}=1 then
   print(" parent stopping");
   quit;                    // 1 returned to parent; it
quits
end;
print( "we made it to", where);
quit;
```

Figure 2. A Sample Obtuse Program

Obtuse is unique in that the mechanisms to support the runtime (threads, contexts, stacks) are all COM objects. Another unique aspect is that Obtuse uses Visual Basic forms to implement user interfaces. These forms can also be marshaled in order to transport their state from one machine to another. Since Obtuse is based on COM, it can be used to manipulate any COM Automation object, which includes all Office applications and ActiveX controls. The DCOM infrastructure supports the remote location and activation of COM objects.

Other features of Obtuse include support for script-based execution, support for multiple threads, runtime strict typing, and a machine-invariant program representation. An Obtuse program can consist of a sequence of expressions with no variables, a series of statements on a set of global variables, or a collection of procedures. Furthermore, an Obtuse program can invoke an Automation object's methods and access its properties at runtime; it is not necessary to "import" or "include" interfaces.

Obtuse does not support compile-time type binding. The type checking in expressions and procedure calls is performed at runtime. However, Obtuse is "strict"; that is, types must match exactly on operations such as comparison or multiplication. Variables are bound to a type on runtime assignment. From that point on, until another assignment occurs, that variable must be type compatible with every operator that is applied to it.

Programs are UNICODE-based and compile to a machine-invariant representation that encodes the source program. That is, the object code can be inverted to recover the original source, including comments.

The paper first presents an overview of ActiveX technology. Then we discuss the Obtuse object model and present an overview of the language. Next, several sample programs are introduced to illustrate the concepts. Also, we present some performance measurements for Obtuse/ActiveX. Finally, we review some of the problems with DCOM based on our experience.

2. ActiveX—COM and DCOM

The two most important aspects of ActiveX for scripting support are its implementation of dynamic method binding and invocation, as well as self-describing types. Dynamic method binding is the technology (Automation) that enables Visual Basic applications to manipulate Office documents, such as spreadsheets or slide presentations. It also enables HTML scripting support (VBScript) in Microsoft's Internet Explorer 3.0.

The dynamic, or late, binding technology enables an object to expose methods and properties for use by other objects. The technology also supports the lookup of method and property names and a mechanism to build and execute a procedure call at runtime. It is a separate set of code from COM.

Self-describing types (or **variants** as they are termed in COM), are the key, underlying representation for data types in the Visual Basic language common to VBScript and VB. In the next sections, we present an overview of COM and DCOM.

2.1 COM – Component Object Model

An "object" in COM typically has a document type, such as .xls, .ppt, .doc. Each document type can have a registered server. For example, winword.exe is the server for *.doc objects. Objects also have a registered application name (e.g. "Microsoft Word Document") and a globally-unique identification number, called a **class id** (CLSID).

The association between a class and its server is maintained in a persistent store called the **registry**. There is one registry per machine and there is currently no "yellow pages" server to support object lookup for distributed services, although one is reputed to be available shortly.

The COM model is language independent; it may have a concrete implementation in a particular language, such as C++, but the relationship between COM and different languages is orthogonal. For example, Obtuse is implemented in C++ but it uses COM objects, which are implemented in Visual Basic, to define its user interface.

A COM object is defined by its support for a collection of interfaces, each is which is tagged by a 128-bit globally unique interface identifier (IID). The interfaces that an object supports can vary over time; and the interfaces need not have any other relationship (such as inheritance). There is only one requirement i.e. **EVERY COM INTERFACE MUST INHERIT FROM THE IUnknown INTERFACE**, which is listed in Figure 3.

```
virtual HRESULT QueryInterface (
    InterfaceID & riid,
    LPVOID * ppvObj)=0;
virtual HRESULT AddRef(void) = 0;
virtual HRESULT Release(void) = 0;
```

Figure 3. COM IUnknown Interface

The power of COM derives from several of the requirements satisfied by the IUnknown implementation. First, QueryInterface must be used to obtain an object handle for an instance variable **x** (as in x.Queryinterface) that supports a particular interface (identified by the InterfaceID argument). If the object **x** does not support the interface, the HRESULT returned indicates an error. In C++, a COM object handle is a "pointer to a pointer to a vTable". The vTable is generated in C++ because the interface is "pure virtual", as are all COM interfaces.

Since every interface is required to inherit from IUnknown, any object handle can be used to retrieve a handle for any interface that the object supports by calling QueryInterface at any time. Further, the object handles are reference counted. QueryInterface increments an object's reference count and so does AddRef. A Release call decrements an object's reference count.

2.2 DCOM – Distributed COM

DCOM extends COM in a number of ways. First, objects can be remotely activated and a handle returned to the activating site. The returned object handle can be used by a program in a location-opaque fashion; that is, the programmer need not be aware of the object's location. Second, DCOM imposes location, security and identity restrictions on COM objects. Each site has total control over who can activate an object, how objects are activated, and with what permissions object servers can execute. Third, DCOM implements reference counting across machine boundaries and garbage collection. Finally, DCOM automatically remotes calls to COM interfaces that are supported by remote objects. For user-defined interfaces, an IDL compiler must be used to generate proxy stubs for the client/server sides. Typically, both stubs are included in a single DLL.

2.2.1 Remote activation

The DCOM method to activate (cause its server to be loaded) a remote object is **CoCreateInstanceEx**. The arguments to the method are the object's class id, a machine name, and a list of interface ids.

Machines are identified using the naming scheme of the network transport layer. By default, all UNC (\\chairpc) and DNS names ("chair.com" or "135.9.19.33") are supported. Object search is restricted to a single machine at present. DCOM has no notion of distributed scope or of distributed search paths.

To optimize network performance, the CoCreate call may specify a list of interface ids. Thus, N object handles can be retrieved in a single round-trip to the server site. Conceptually, this is analogous at runtime to the "import java.lang.*" convention in Java, which can be used to import all of the classes in a package at compile-time.

2.2.2 Access control

Access to objects can be regulated under program control using the NT security API; however for most Obtuse users, the utility program **dcomcnfg** is the point of control. This program lists the application objects that are "registered" on a particular machine. The **Location**, **Security**, and **Identity** of each object can be separately controlled. The Location options are "run here" or "run there". The latter option supports forwarding a requested activation from one computer to another. The Security option supports editing the access control lists (ACLs) for activation, access and configuration. NT provides very fine-grained access control so that individual users, or groups, can be specified.

The Identity option designates the protection domain in which a server is executed. The choices are the domain of the interactive user, the launching user, a particular user, or a system service. For example, the "particular user" option can be used to solve the "game accounting" problem, which is to let a user run a game program that can write its list of winners to a file that is not accessible to one of the players. The appropriate protection domains can be created by having one

DCOM object (player's domain) to play the game communicating with another DCOM object (game's domain) to record the scores.

2.2.3 Reference counting

As we discussed in Section 2.1, COM defines a mechanism to reference count object handles. If a program fails, any cross-process links must be broken to properly release objects. Similarly for DCOM, the system must account for inter-machine links and must break links when processes terminate or fail. For distributed systems, there are also the possibilities of node crashes and communication outages. The DCOM implementation addresses these problems.

2.2.4 Remote procedure call

DCOM automatically remotes inter-node calls on COM interfaces. The arguments are marshaled through the normal remote procedure call (RPC) mechanism. RPC on user-defined interfaces requires the use of the IDL compiler to generate client- and server-side proxy stubs.

The automation interface (IDispatch) can be used to "late bind" a procedure call; that is, a program can build a procedure call at runtime. The automation interface provides the object-access infrastructure for any scripting language, such as Obtuse, VBScript, JavaScript or AppleScript.

COM supports the registration of type libraries that describe an object's properties and methods (also argument lists and return values). As a property example, a button object might have BackgroundColor and Text properties, which could be accessed or modified remotely using Obtuse. The IDispatch interface includes methods to "query" for the id of a method or property name and then to "invoke" that method or "access" that property. The Automation runtime builds the argument list in a format that is compatible with the target language and handles the call/return processing.

2.2.5 Variant data

Another aspect of the automation solution is a "universal" data type termed a **variant**. A variant is a "union" of about 40 different base types that also includes arrays of those types, and arrays of arrays. An array can be created with homogeneous elements of a particular type or with variant-type elements, each of which can be of any type.

IDispatch and IUnknown object handles are two of the possible variant-record base types. Since IDispatch is one of the "builtin" COM interfaces, it is remoted automatically by DCOM. In Obtuse, all argument lists to procedures, return values, and property values are encoded as variant data.

3. Obtuse Language Overview

The Obtuse system consists of a compiler and an interpreter, and a collection of COM objects. The compiler's output is a UNICODE text string that encodes the source program, including comments. The executable can be inverted to produce the original source program. Thus, after a program is initially compiled, there is only one representation, which can be used for both execution and symbolic debugging.

Obtuse supports only one data type (variant), so a variable declaration is just a list of identifiers. The type is implicit. A form of type checking is supported based on the notion of assignment-typing. Basically, every assignment statement binds a new type to an identifier as well as a new value. Expression evaluation is type checked at runtime. There is no implicit conversion as in VB; that is, type checking is "strict".

In Obtuse, the "object" built-in function maps a registered object name at a particular machine to an object reference. For example, the function call **object("Bob.Thread", "foo.univ.edu")** would check the registry on the designated machine and then load the server if necessary.

Once an object reference is obtained, the program can manipulate the properties of that object or invoke its methods. Assignment of object references copies the reference, not the value, even if the assignment crosses machine boundaries. The DCOM reference counting infrastructure tracks each copy. For assignment of other variant values, including arrays, Obtuse copies the value. The rule is simple: sharing can only be accomplished through COM objects.

Obtuse implements a common set of statements such as **if**, **loop**, **for**, **case,** in addition to variable and method declarations. Pointer, structure and class declarations are not supported. A qualified reference can be used to access an object's properties. Since an object's methods are dynamically bound using the IDispatch automation interface, the compiler cannot perform checking for undefined names or mismatched argument lists. To facilitate some checking, calls to Obtuse procedures are delineated with the traditional "( )" and calls to an object's methods use "{ }".

As mentioned earlier, Obtuse programs are encoded as UNICODE strings. Sufficient information is retained in the encoding to invert the object code to the source. The opcodes were designed to use a character encoding

so that program fragments could be embedded in documents, sent as mail messages, or be applied as drag-and-drop operators on user-interface objects. Figure 4 lists several example encodings. The blank, tab, and new-line opcodes are no-operations.

The " opcode designates a constant. Constants are translated at runtime so the opcode includes a type designator, the length of the string, and the text constant. This is not very efficient but it does avoid representation issues, such as for floating-point numbers. Small integers are encoded as individual opcodes. The opcode design also took into account the requirements for the next version of Obtuse in which type modules, such as Complex numbers, could be called upon to parse their own constant representation.

Code Fragment	Encoding
print(3/24/76);	"D073/24/76 p0 q
print(3+45);	3 "L0245 + p0 q
if a>3 **then** y := 6;	Y3 L1 3 : > I005
	6 S2 J015
elsif a>2 **then** y := 8;	Y4 L1 2 : > I005
	8 S2 J007
else y := 9; **end**;	Y5 9 S2 Y6

OpCode	Key
"	Load Constant
Y	Syntax Marker
L	Load Variable
S	Store Variable
:	Compare
I/J	Forward Jumps

Figure 4. Program Encoding Example

The Y opcodes encode the syntax of the source program. Even the comments in the source are encoded, but the comment opcode is treated as a no-op at runtime. The compiler attempts to generate code for branch instructions so that syntax markers are not included in loops.

4. Obtuse Object Model

The initial Obtuse implementation supports FORM, FILE, MUTEX, THREAD, CONTEXT, and STACK objects. FORM objects are Visual Basic forms, which can contain any VB control. Each FORM object represents one VB form. Since there are hundreds of different VB controls, the Obtuse user interface model has a broad range of capabilities. As a result, forms can be constructed as the user interface for almost any application.

The FORM interface is implemented as a Visual Basic program. VB supports the creation of programs that support COM interfaces (particularly IDispatch, the Application Automation interface). As a result, these programs, can be activated remotely using DCOM. We implemented (in VB) a form-server object that supports Form, Item, Save and Restore methods. The Form and Item functions return object references to a form or to any of the controls on that form. Once an object reference is obtained, Obtuse can set or retrieve the properties of a form or control. For example, the "value" property of a scroll bar is a numeric quantity that can be used to get/set the thumb position.

In the current prototype, a programmer constructs a user interface with a VB program called GenForm, which is part of the Obtuse system. When GenForm is executed, it writes a file that contains a text array constant that "defines" a form. The array constant is then inserted into an Obtuse program as a "resource". The Restore method causes the VB form-server object to display the previously-saved "look". A VB form is encoded/decoded by Save/Restore as a text array. Figure 5 illustrates the encoding of the Roving Poller form that was displayed in Figure 1.

```
[12345, 12, 3, 15, 4, 5535, 1, 2145,5,
16776960, 2, "Roving Poller", 23456,
14,3, 72, 4, 144, 0, 89, 1, 41, 5,
-2147483633, 2, "OK", 45678, 12, 3, 0,
4, 88, 0, 125, 5, -2147483633, 2,
"Enter your comments below:",
56789, 12, 3, 16, 4, 0, 0, 361, 1, 49, 6,
""]
```

Figure 5. Array Constant for a VB Form

To save space when creating a new form, the GenForm program only saves the differences between a canonical set of control values and those specified by the user. For example, the "top" and "left" properties are almost always changed; the "visible" property is rarely changed. Since VB has a large number of properties for each control, this convention saves considerable space.

Obtuse has the unusual property that the mechanisms of the language implementation are objects, in fact DCOM objects. Remember that a DCOM object can be activated on any machine. The Obtuse interpreter is only required to run Obtuse code, not remote objects. This is one of the main differences with Obliq and other distributed application systems, which require an instance of their interpreter at each node.

A FILE object supports I/O on files and directories anywhere in the Internet. Interestingly, DCOM can pass a file handle from one machine to another and the handle retains its validity. This is not possible with NT, the host operating system. Since activating a FILE object is necessary to access files and since DCOM implements per machine and per object access controls, the user has full control over the safety of the system.

A MUTEX object is used to implement critical section synchronization for shared variables or resources. The supported methods are Enter and Leave.

The Obtuse object model takes unique advantage of DCOM's capabilities. First, thread and context (a program's global variables) objects can be created on any DCOM machine on the Internet so that a thread on one machine can opaquely access variables on any other machine's context. Figure 6 lists the attributes of the three Obtuse program objects – Thread, Context, and Stack.

A context can be shared among any number of threads, local or remote. For example, a master debug console can easily be constructed to monitor the modification of contexts located all over the world. Finally, a thread can migrate by simply forking its state to a new machine and killing the parent thread. Since a context is one of the arguments to the Fork method, the new thread can be created with its own copy of the parent's context or it can share the parent's context. All object references are marshaled properly by DCOM on inter-machine transfers so that programs remain completely location opaque.

STACK objects are always co-located with their thread objects; however, they still are DCOM objects. There is no requirement that execution be "procedure based". The interpreter can evaluate formulas with only a stack and a code string. When a thread is marshaled to be transferred to another machine, the stack content, including return addresses, is converted to a portable format.

In the COM model, objects are normally created by server front-ends called class factories. The separation of request and creation on a per-type basis provides a way to create many different types of servers for each object type. Remember that in COM an object is defined by the interfaces that it supports, not by its data structures or algorithms.

4.1 The thread object

In the current implementation, a THREAD object contains a code string, an IDispatch object handle to a context object, two object handles to a stack object, and type information (used by IDispatch, described later). The IOperator interface defines methods such as Add and Subtract; the IProcedure interface defines methods such as Frame and Return (used for procedure call/return). The latter interface may be omitted for calculations that do not involve procedure calls.

	DATA	OBJECT INTERFACES
THREAD	Code string Context Stack Type Info	Com.IUnknown Com.IDispatch IThread
CONTEXT	Array Variants Type Info Persist flag	Com.IUnknown Com.IDispatch IObject
STACK	TopOfStack ProcFrameIndex FrameTopStack Array of Frames Array Variants	Com.IUnknown Com.IDispatch IOperator IProcedure

Figure 6. Obtuse Program Objects

When a thread starts execution, it binds to an IObject handle. For efficiency (since a context holds global variables), the IObject interface was compiled by the IDL compiler to generate proxy stubs. As a result, references to a remote context, must pass through a proxy DLL, which must be registered at that site. Accessing global variables using the IObject interface is much faster than using IDispatch.

Figure 7 lists the IThread interface, which contains methods for creating a thread, marshaling its state, and controlling its execution. Migrating a thread or object depends on support for marshaling its state. In theory, any system, such as C++ or Java, could support migration.

4.2 The context object

A context object is a vector of global variables. Since all variables in Obtuse are represented using the variant data type, a context is just an array of variants. Further, since an array of variants is also a variant type, a context is marshaled automatically by DCOM when passed from one machine to another.

METHOD	ARGUMENTS	USE
Open	Code string	Start a thread with a default context
OpenEx	Code string Initial pc value Initial context	Restart a thread from a saved state

	State to restore Suspend flag	
Fork	Thread object Context object	Clone the current thread and context
Join	Timeout value	Wait for a thread to terminate
Suspend		Stop a thread
Resume		Start a thread
Sleep	Delay value	Timed delay
Code		Retrieve code string
Context		Retrieve context handle
Stack		Retrieve stack handle

Figure 7. The IThread Interface

The IObject interface, which is listed in Figure 8, describes the methods to store and retrieve Obtuse variables. The Get/Put methods access simple variables; GetIndex/PutIndex access arrays. Since all Obtuse variable locations are the same size, variable addresses are just indices (e.g. 0,1,2 etc.).

Array access is implemented by passing the entire subscript list as an argument. This approach is more efficient for remote access than evaluating one subscript at a time. In Obtuse, array assignment is "by value". The only way to generate a "reference" in Obtuse is by creating a COM object.

The context class interface contains a number of helper functions (save, restore, persist, sweep) that are intended only for local use. The "save" and "restore" functions are used to clone a context. The "persist" helper function toggles a flag that indicates whether a context should be retained after its thread terminates. This option is useful for debugging and also for writing programs that inter-operate by passing contexts back and forth. Used in this way, a context is somewhat like a COMMON block in FORTRAN.

METHOD	ARGUMENTS	USE
Get	Index	Access a variable
GetIndex	Index Subscript list	X[a, b, c]
Put	Index	Store a variable
PutIndex	Index Subscript list	X[a, b, c] = 3

Figure 8. The IObject Interface

The "sweep" helper was introduced after we discovered that many of our early programs were leaving objects scattered all over the network. The program in Figure 1 illustrates the problem. A remote context is created and then the local context is "cloned" into it. However, the new context now has a reference to itself, since its handle was in the original context. As a result of this circular reference, DCOM never called the destructor for the context when the thread terminated. The "sweep" method addresses the problem by clearing all object handles in a context when its thread terminates.

4.3 The stack object

As mentioned earlier, there is a one-to-one relationship between a stack and a thread. By design, a stack can never contain a reference to itself so circular references are prevented. Stacks and threads are always co-located for efficiency. In Obtuse, a stack is a vector of variant values and may also have an associated vector of frames (if procedures are used by the code fragment).

This design is somewhat unconventional in that most systems embed the call chain within the evaluation stack. The disadvantage is that the chain typically uses pointers, which we avoid by inverting the list. As a result, the frame stack is a separate array. When a thread migrates, there are no restrictions on its state. It can be arbitrarily nested within procedure calls.

Each call frame contains an argument count for the procedure, a count of local variables, the evaluation stack index for the previous frame, and the index of the call point in the thread's code string. Another advantage of inverting the frame stack is that arguments and locals are adjacent so indexing is trivial.

Figure 9 lists the IOperator and IProcedure interfaces. The stack class includes two helper functions: Save and Restore. The "Save" procedure produces an array of variants that represents the "state" of the stack, including procedure nesting. The "Restore" procedure returns a stack to a previous state. Since program state is a first-class object in Obtuse, it should be possible to support fault-tolerant algorithms through various checkpointing schemes; however, we have not explored this idea further.

IOperator	
Add	3 + 4
Subtract	3 - 4
Multiply	3 * 4
Divide	3 / 4
Compare	< <= = >= >
Mod	3 % 4
Invert	- or !
And	3 & 4

Or	3 | 4
Load	Variant←constant string
Print	String←variant
Push	Stack←variant
Pop	Variant←stack
IProcedure	
Frame	Call procedure
Return	Return from procedure
Get	Get local/argument
GetIndex	Get local array
Put	Store local/argument
PutIndex	Store local array

Figure 9. The IOperator/IProcedure Interfaces

4.4 Type information

The power of the Obtuse object model derives from DCOM, particularly when combined with IDispatch. In this section, we illustrate how a dynamic procedure call can be accomplished. As is illustrated in Figure 1, the "object" statement in Obtuse can be used to create an object on any machine. In COM, an object handle is created by asking if the object supports a particular interface. Obtuse always queries for the IDispatch interface.

Thus, the "me" variable in Figure 1 is a variant record with a value that is an object handle of type IDispatch. The code "me.Fork{a, b}" translates to interpreter byte codes that indicate a method invocation. When the Obtuse interpreter encounters an "invoke" opcode (actually properties work the same way), it first has to bind the method name (Fork) to a method id code. Automation does not support a call by name option. As a result, it takes two round-trip calls to the server per method call.

After getting the method id, the interpreter calls the "Invoke" method in the IDispatch interface of the "me" object. The arguments to "Invoke" are a vector of variants (the argument list) and the method id. The return value from "Invoke" is a variant that encodes the result of the "Fork" call.

The remaining part of the puzzle is a discussion of how IDispatch actually constructs a method call to "Fork" in whatever language "Fork" is implemented, and with the appropriate calling convention. Figure 10 illustrates the solution used in Obtuse.

```
static PARAMDATA
        rgpdataCBobContextPersist[] =
{
  { "onOff", VT_LONG }
};

static METHODDATA rgmdataCBobContext[] =
{ // void CBobContext::Persist(long onOff)
  {
   "Persist",
   rgpdataCBobContextPersist,
   IDMEMBER_CBOBCONTEXT_PERSIST,
   IMETH_CBOBCONTEXT_PERSIST,
   CC_STDCALL,
   DIM(rgpdataCBobContextPersist),
   DISPATCH_METHOD,
   VT_VOID
  },
};

static INTERFACEDATA g_idataCBobContext=
{
  rgmdataCBobContext,
  DIM(rgmdataCBobContext)
};
```

Figure 10. Encoding an IDispatch Interface

The data structure encodes the "type information" referred to in Figure 6 for thread and context objects. The "interface data" structure contains a count of the number of methods, and pointers to method descriptors. Each method descriptor contains the name of the method, a pointer to a vector of parameter descriptors, a method id number, an index into the vTable for the class, a count of the number of parameters, a code to indicate the calling convention, and the type of the return value.

Thus, for the "me.Fork{}" example, the arguments (encoded as an array of variants) are converted to the parameter types specified, pushed on the stack in a calling-convention and language-specific way, then the vTable index is used to make the call. The return value is removed from the stack (or registers) and is encoded as a variant. The Obtuse interpreter then pushes the return value on its stack and execution continues.

5. Sample Programs

We have identified three classes of distributed applications that can be programmed using Obtuse – Migratory, Synchronized, and Cooperative. A **migratory** application moves object state from one machine to another. This may involve moving all of a program, or only a part, such as the user interface. Implementing routing slips for documents is an example of a migratory application. A **synchronized** application is one in which the actions at one site are duplicated at another. For example, a debug console might be

synchronized to the program or UI state of a distributed application. Finally, a **cooperative** application requires that multiple sites participate to solve problems. Decision-making tasks, such as preparing budgets for instance, are usually accomplished in a cooperative fashion.

scroll bars on a second machine. The objective was to increase the number (NBARS) of scroll bars, and then to observe the impact on responsiveness.

The program actually does not scale very well, but the problem is with the algorithm, not DCOM. Distributing a signal to a large number of recipients should not be

```
var resource;     //set from a VB form
var a, b, c, where, me, thread, context;

resource :=
[12345,12,3,-15,4,5535,1,2145,5,16776960,2,
"Roving Poller",23456,14,3,72,4,144,0,89,1,41,5,-2147483633,2,
"OK",45678,12,3,0,4,88,0,125,5,-2147483633,2,
"Enter your comments below:",56789,12,3,16,4,0,0,361,1,49,6,
""];
foreach where in argv do                //iterate command line arguments
  me := self;                           //a reference to the running thread
  thread := object("Bob.Thread", where);   //create a remote thread
  context := object("Bob.Context", where); //create a remote context
  if me.Fork{thread, context}=1 then    //duplicate myself at "where"
    quit;                               //1 returned to parent thread; it quits
  end;                                      //-------child thread starts here
  a := object("Bob.Form", "");          //create a VB form at the child site
  b := a.Restore{resource};             //method call to VB form server
  c := a.Item{"button0"};               //get object reference to the button
  loop
    if c.tag = "1" then exit; end;      //delay until a button click
  end;
  resource := a.Save{};                 //save the current look and content of form
end;                                        //loop until all the sites have been visited
quit;
```

Figure 11. A Migratory Application – Roving Poller

Figure 11 lists the code for the Roving Poller example discussed in Section 1. It is an example of a migratory application. The program transfers itself to every machine in a list, which is specified on the command line, so that each user can enter comments in an edit control. The completed form, together with the accumulated comments, is displayed at the last machine in the list. A return-to-sender convention could be implemented by placing the name of the originating machine last in the argument list.

The second example, which is listed in Figure 12, is a simple, synchronized application that was written as a UI performance test. The idea was to synchronize a scroll bar on one machine with an arbitrary number of

performed with a simple **for** loop, but rather with a distribution hierarchy.

The final example, which is listed in Figures 13 and 14, implements a cooperative application that supports a common decision-making task performed by three co-workers; that is, deciding where to go to lunch in a timely, and fair, fashion. When the application is initiated, it displays a form containing three edit controls at each site. The participants can type in their choice for lunch and can observe, but not modify, the other choices. Each user can change their mind arbitrarily, but at the instant that a majority has agreed on a choice, input is frozen and the consensus is displayed for all to see.

```
var a, b, c, e, old, i, NBARS;
var aa, cc;
var resource;
resource :=
[12345,14,3,6120,4,6600,0,2880,1,1110,5,8454143,2,
"Scroll Test",23456,4,5,8454016,89013,8,3,16,4,16,0,153];
NBARS := 9;
aa := [0];                          //create an array dynamically
for i :=0 to NBARS-2 do             // should probably be a function to do this
  aa := aa & [0];
end;
cc := aa;
a := object("Bob.Form");                  //create the master scroll bar
for i:=0 to NBARS-1 do
  aa[i] := object("Bob.Form, "a-bobc-1"); //create N remote scrollbars
  b := aa[i];
  c := b.Restore{resource};
  c := b.Form{};                    //get an object reference to the form
  c.Top := (i%8)*1000;              //place the scroll bars in a grid pattern
  c.Left := (i/8)*4000;
  cc[i] := b.Item{"hscroll0"};      //object reference to each scroll bar
end;
b := a.Restore{resource};
c := a.Item{"hscroll0"};
old := c.value;
loop
  e := c.value;
  if (e > 30000) then exit; end;    //exit when thumb moved to far right
  if e != old then                  //wait for a state change
    for i:=0 to NBARS-1 do          //update all the other scroll bars
      b := cc[i];
      b.value := e;
    end;
    old := e;
  end;
end;
quit;
```

Figure 12. A Synchronized Application – Master/Slave Scroll Bars

6. Performance Measures

Obtuse implements late binding of procedure calls and property access by using the Automation IDispatch technology. Obtuse marshals program and user interface state by encoding values in variant arrays. There is the question of what penalty is paid for the additional complexity.

We conducted performance tests in order to quantify some of the costs. The tests were conducted on two 166 Mhz Pentiums, which were on the same 10mb Ethernet segment, and which were running Windows NT 4.0. Table 1 lists the results of the tests. Each test program was run several times to verify that the results were stable and each test loop was repeated 100 to 10,000 times, depending on the amount of time involved. The test programs are listed at the Obtuse web site: obtuse.cs.olemiss.edu.

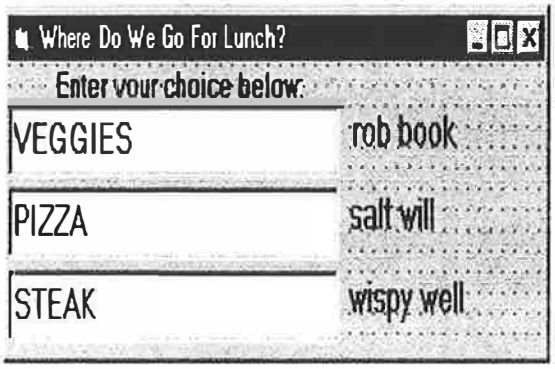


Figure 13. The Eat-Lunch User Interface

Obtuse has no program library to provide timing functions so the interpreter was modified such that a reference to the first global variable in a context returned the current time in milliseconds. Thus, two references to the same variable could be used as a timing function.

The test results require some explanation. First, there is a distinct time difference between variable access/function calls and invoking a method on an Automation object. The latter operation is slower because Obtuse must do a symbol table lookup to bind the method name to a method index. Access to Visual Basic properties or methods takes even longer, up to 1.6ms. Since there is no difference in the Obtuse runtime code between accessing a VB object and an Obtuse object (like Event), we can assume that the VB runtime contributes to the factor of 25 performance decrease.

The remote variable access only takes 4ms, versus 11ms for the function call, because context object access is routed though a proxy DLL on each machine whereas access to other Obtuse objects, such as EVENT or FILE, must use the IDispatch infrastructure. IDispatch requires two separate calls to the remote node for each method call: one call to bind the name and one to make the call.

Table 1. Obtuse Performance Measurements

Function	Timing	Software Layers
Global variable (in a Context)	0.016 ms	1:in-process proc call 2: array access
Local Obtuse function call	0.025	1:in-process proc call 2: variant copies
Local COM object function call	0.069	1:in-process call 2:COM runtime 3:IDispatch runtime
Local Visual Basic COM object function call	1.543	1:cross-process call 2:COM runtime 3:IDispatch runtime 4:VB runtime
Local VB property reference	1.610	1:cross-process call 2:COM runtime 3:IDispatch runtime 4:VB runtime
Local Clone task (one cycle)	4.400	1:variant copies(lots) 2:memory allocation 3:new OS thread 4:kill OS thread 5:memory free (all in process)
Remote global variable access (in a Context)	4.600	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:net transport
Remote COM object function call	11.020	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2)

Remote VB property reference	11.820	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:VB runtime
Remote Visual Basic COM object function call	12.000	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:VB runtime
Local Restore and Save of VB Form	43.760	1:cross-process call 2:COM runtime 3:IDispatch runtime 4:VB runtime
Remote Restore and Save of VB Form	81.100	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:VB runtime
Remote Clone task and then remote clone back to the source (one cycle)	266.400	1:variant copies(lots) 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:new OS thread 7:kill OS thread 8:memory free

The final set of tests involved cloning a task to a target machine and then cloning that task back to the source. This provides an indication of the costs to migrate a process from one machine to another. The migration test was performed without also moving a user interface. The local test (in which the target and source were the same machine) took 4.4ms and the remote test took 266ms

7. Observations

The Obtuse system was designed and implemented by the author in eight weeks as part of a summer research appointment at Microsoft Corporation. As a result, both the language and runtime lack a number of features that can be found in more mature languages such as C++ or Java. Nevertheless, there are some lessons that can be shared based on the experience to date.

First, migrating a thread or object depends on support for marshaling its state. In theory, any system, such as C++ or Java, could support migration, and probably should.

The ability to transmit a user interface's state from one machine to another is also a useful capability. Visual Basic and its competitors could easily be extended to support persistence. The VB property model even lends itself to simple encoding strategies.

We found that support for marshaling the state of forms was essential to the implementation of migratory applications. Most Office applications, for example, marshal (or save) the current user interface state on exit, and restore it on startup. Java has provided entry points for marshaling UI state with the Applet methods init/destroy and start/stop. JDK 1.1 has additional support for serialization. Client-side Java has been joined by server-side Java. The next evolution should be mobile Java.

The most difficult implementation problems involved reference counting, which resulted in objects that never got released. For example, in testing the Roving Poller example, it was discovered that context objects were being activated, but never deleted, on machines all over our building. The problem turned out to be a circular reference; that is, the context objects had references to themselves. The problem was addressed by checking for circular references in both the stack and context object of a terminated thread. However, this approach would not handle indirect recursion through other context objects.

DCOM provides an excellent infrastructure for writing distributed applications. However, its OLE legacy is the source of a number of serious problems. The first, and most serious, problem is the OLE registry, which is used to store app/server/classId associations and access control information. As a result, Windows NT has two object information systems—the file system and the registry. Even worse, the DCOM configuration information can only be manipulated by the system administrator. Also, the tools that support changes to the registry are neither user friendly nor fault tolerant. As a result, for multi-user systems, such as all the machines in the CS department, no students can create objects. This situation must be resolved before DCOM can find wide acceptance.

A less severe problem is the current specification of IDispatch/Variants, which were designed to support Visual Basic, and were later modified slightly to support

Visual Basic for Applications. The design is not well-suited for supporting distributed applications. The concepts are correct, but the requirements have changed. As a result, the design needs to be upgraded. For example, the Variant value types are not extensible; variants only support what was required to implement Visual Basic. There is no reason to require reference counting for value-based types, such as complex numbers or x-y coordinates.

8. Related Work

Obliq depends on Modula-3 for its runtime support; Obtuse depends on COM/DCOM. Bharat's Visual Obliq also supports migrating user interfaces. Obtuse is unique in its integration of the COM object model in its design and in the support for persistent VB objects. Hirano's [6] HORB system is a Java superset that can be used to write distributed applications. Gray[7] has implemented a transportable agent system, Agent Tcl, and maintains an extensive "related work" site[8].

9. Acknowledgements

At Microsoft, Tony Williams conceived of, and initiated, the Obliq/DCOM summer project, which was supported by Nat Brown and Dennis Adler.

10. Availability

The Obtuse system is available for experimentation at the web site www.obtuse.cs.olemiss.edu.

References

1) Cardelli, L., Obliq: A language with distributed scope. Report No. 122, Digital Equipment Corporation, Systems Research Center, (1994).

2) Krishna Bharat and Luca Cardelli, Migratory Applications, Proceedings of ACM Symposium on User Interface Software and Technology '95, Pittsburgh, PA, (Nov 1995). http://www.cc.gatech.edu/gvu/people/Phd/Krishna/VO/Migration.html

3) Distributed COM, Microsoft Corporation (1996).http://www.microsoft.com/windows/common/aa2399.htm

4) The Component Object Model Specification, Microsoft Corporation (1996). http://www.microsoft.com/oledev/olecom/title.htm

5) Brown, Nat, and Kindel, Charlie. Distributed Component Object Model Protocol -- DCOM/1.0, (1996). http://ds.internic.net/internet-drafts/draft-brown-dcom-v1-spec-01.txt.

6) Hirano, Satoshi, The HORB System, (1996). http://ring.etl.go.jp/openlab/horb/

7) Gray, R.S. et al,. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Department of Computer Science, Dartmouth College, 1996.

8) Related Work, http://www.cs.dartmouth.edu/~agent/

```
var resource, a, b, c, d, e, f, g, h, i;
resource :=
[12345,12,3,-15,4,5535,1,2085,5,8454016,2,
"Where Do We Go For Lunch?",45678,12,2,
"Enter your choice below:",4,32,3,0,0,117,5,-2147483633,45679,14,2,
"   ",4,234,3,16,0,9,1,25,5,-2147483633,45680,12,2,
"",4,232,3,48,0,3,5,-2147483633,45681,12,2,
"",4,232,3,80,0,3,5,-2147483633,56789,12,3,16,4,0,0,225,1,25,6,
"",56790,12,3,48,4,0,0,225,1,25,6,
"",56791,12,3,80,4,0,0,225,1,25,6,""];
a := [ ["daddyo", "rob book"],  ["monroe", "salt will"],
    ["a-bobc-1", "wispy well"]];
g := [ ["label1", "text0"], ["label2", "text1"], ["label3", "text2"]];
b := [0,0,0];    i := [0,0,0];
for c:=0 to 2 do
  b[c] := object("Bob.Form", a[c,0]);          //create the form at each machine
  d := b[c];
  e := d.Restore{resource};
  e := d.Item{g[c,0]};                          //get an object reference to each label control
  e.caption := a[c,1];                          //set the "caption" property to the user's name
  for h:=0 to 2 do                              //lock all the edit fields except the user's
    f := d.Item{g[h,1]};
    f.locked := h!=c;
  end;
end;
loop
  for c:=0 to 2 do
    d := b[c];
    f := d.Item{g[c,1]};                        //retrieve the choices of the other users
    i[c] := f.text;
    for h:=0 to 2 do                            //update my controls to display their latest
      if h!=c then
        d := b[h];
        f := d.Item{g[c,1]};
        f.text := i[c];
      end;
    end;
  end;                                          //as soon as 2-of-3 match, let majority rule
  if (i[0]!="") & (i[0]=i[1]) then i[2] := i[1]; h := 2; exit; end;
  if (i[1]!="") & (i[1]=i[2]) then i[0] := i[1]; h := 0; exit; end;
  if (i[0]!="") & (i[0]=i[2]) then i[1] := i[0]; h := 1; exit; end;
end;
for c:=0 to 2 do                                //lock every control, display consensus everywhere
  d := b[c];
  f := d.Item{g[c,1]};
  f.locked := true;
  f.text := i[c];
  f := d.item{g[h,1]};
  f.text := i[c];
end;
```

Figure 14. Where-To-Eat-Lunch Application

Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance *

P. Narasimhan, L. E. Moser, P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
priya@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

The Eternal system is a CORBA 2.0-compliant system that provides, in addition to the location transparency and the interoperability inherent in the CORBA standard, support for replicated objects and thus fault tolerance. Eternal exploits the Internet Inter-ORB Protocol (IIOP) interface to ''attach'' itself transparently to objects operating over a commercial CORBA Object Request Broker (ORB). The Eternal Interceptor captures the IIOP system calls of the objects, and the Eternal Replication Manager maps these system calls onto a reliable totally ordered multicast group communication system. No modification to the internal structure of the ORB is necessary, and fault tolerance is provided in a manner that is transparent to both the application and the ORB.

1 Introduction

Distributed systems consist of clusters of computers that are capable of both functioning autonomously and cooperating harmoniously to achieve a particular task. The integration of an object-oriented paradigm with a distributed computing platform yields a framework in which objects are distributed across the system. Objects invoke other objects, or are themselves invoked, to provide services to the application.

The Object Management Group (OMG) has established the Common Object Request Broker Architecture (CORBA) [12, 14, 15, 17, 18], which is a standard for communications middleware that defines interfaces to distributed objects and that provides mechanisms for communicating operations to objects by means of messages. The key component of this architecture is the Object Request Broker (ORB), which handles requests to, and responses from, the objects in the distributed system.

*Research supported in part by DARPA grant N00174-95-K-0083 and by Sun Microsystems and Rockwell International Science Center through the State of California MICRO Program grants 96-051 and 96-052.

Unfortunately, the current CORBA standard makes no provision for fault tolerance, which has led to research aimed at making CORBA-based applications reliable. One approach has been to build the fault-tolerance capabilities into the ORB itself [16], as in Electra [8, 9] and in Orbix+Isis [5]. Another approach, adopted in the OpenDREAMS project [3], advocates that reliability be provided as part of the suite of object services available to the ORB. While the former approach makes the fault tolerance transparent to the application, it also involves considerable modification to the CORBA implementation to enable the ORB to take advantage of a multicast group communication system underneath it. On the other hand, the latter approach simply adds an object group service on top of an unmodified ORB, and uses no underlying multicast group communication system, thereby making the system interoperable and portable, but with the fault tolerance visible to the application programmer.

The Eternal system that we are developing provides fault tolerance transparently to the application using CORBA, without modification to the ORB. The mechanisms for achieving reliability are hidden from the application programmmer, and concern only the system developer. The Eternal system can utilize any commercial implementation of the CORBA 2.0 standard. Although Eternal is layered over a multicast group communication system, the vendor's ORB does not need to be altered to utilize the fault tolerance that Eternal provides. Furthermore, the system is designed to enable objects running over different ORBs to interact with each other.

The Eternal system exploits the services provided by the Totem multicast group communication system [1, 6, 11] to maintain the consistency of the replicas that are employed for fault tolerance. However, since Eternal only deals with interfaces of objects and of the ORB, any multicast group communication system, with an interface and guarantees similar to those of Totem, can alternatively be used.

The structure of the Eternal system is shown in Figure 1. In this paper, we focus on the Interceptor, which ''catches''

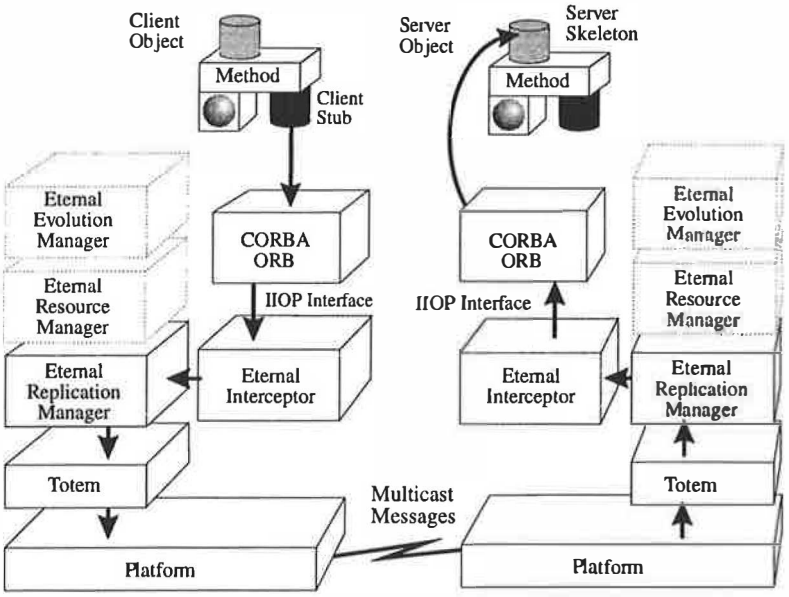


Figure 1: Structure of the Eternal system.

the system calls made by the ORB to TCP/IP, and also on the relevant part of the Replication Manager, which diverts the calls to Totem. In addition, Eternal supports the evolution of a system by exploiting the replication of objects to perform live upgrades of objects and their interfaces. Resource management is also provided for the creation, placement, and distribution of objects.

2 CORBA and the IIOP Interface

The CORBA standard specifies an interface for each distributed object. This interface is written in the declarative syntax of the OMG Interface Definition Language (IDL). The language-specific implementation of a server object is hidden from client objects that require the service provided by the server object; the server object can be invoked only through its interface.

Invocations of objects and responses from invoked objects are handled through the ORB, which acts as the intermediary or "communication bus" for all of the interactions between the distributed objects in the system. At a client object, a stub, generated by the IDL compiler, receives the request, marshals the call into the format appropriate to the request, and passes it to the ORB. At the server object, a language-specific mapping of the IDL specification, a skeleton, unmarshals the parameters of the call and performs any additional processing to invoke the appropriate method. The results of the operation are returned to the client object via the ORB.

CORBA provides location transparency, meaning that the client objects convey their requests only to their ORBs, which then undertake the task of locating a suitable server object and then dispatching the request to it. Thus, a client object need not be aware of the location of a server object since the ORB has access to this information. Every CORBA object is identified by an object reference, which is assigned to it by the ORB at the time the object is created. Client objects associate object references with their requests to enable the ORB to route their requests to the appropriate destinations.

The interoperability of CORBA arises in the context of communication between heterogeneous ORBs. Every CORBA 2.0-compliant ORB is equipped with the ability to communicate using the Internet Inter-ORB Protocol (IIOP) [10, 12], which ensures that objects running over different ORBs can interwork when they use the IIOP interface. Only the ORB hosting an object needs to know the details of the object, while other ORBs that wish to interact with the object need only be able to address it. Every object is assigned an Interoperable Object Reference (IOR) for this purpose.

The General Inter-ORB Protocol (GIOP) is a general set of specifications that enable the messages of the ORB to be mapped onto any connection-oriented medium that meets a minimal set of assumptions (reliable, byte stream-oriented, loss-of-connection notification). The Internet Inter-Orb Protocol (IIOP) is GIOP with the messages transported by TCP/IP. By sending IIOP messages over TCP/IP, the ORBs can use the Internet as the backbone for their communi-

cation. Server objects, that use IIOP to interact with their client objects in an environment of heterogeneous ORBs, publish their references in the form of IIOP IOR profiles.

The primary motivation for the use of IIOP is that all CORBA 2.0-compliant implementations can use this simple generic interface, irrespective of the internal details of the vendor's ORB, and the platform on which the ORB operates. A number of commercial ORBs now provide IIOP as their native protocol, since an increasing number of CORBA applications require interoperability over different platforms and the ability to operate over the Internet.

3 The Eternal System

The Eternal system is designed to work with any commercial off-the-shelf CORBA 2.0-compliant ORB with no modification whatsoever to the ORB. Moreover, the fault tolerance is provided in a manner that is transparent to the application objects.

Since the underlying fault tolerance capabilities are hidden from the application, the application programmer does not need to worry about the difficult issues of asynchrony, replica consistency, concurrency, and the handling of faults. The Eternal system replicates and distributes the application objects across the system, and allows the programmers to write the application as if it were a sequential program to be run on a single machine.

Fault tolerance is provided by replication [7] of both client and server objects across the distributed system. As shown in Figure 1, Eternal exploits the reliable totally ordered message delivery of the underlying Totem system to ensure replica consistency in the presence of faults. In addition, mechanisms are provided to detect and suppress duplicate operations and to support nested operations [13].

4 Group Communication Models

4.1 Process Groups

An increasing number of distributed applications are structured as collections of processes that interact or cooperate to accomplish a particular task. Such a collection of processes is called a process group and can be considered abstractly as a single unit, as shown in Figure 2. A process group may reside entirely within a single processor, or may span several processors.

A process group is characterized by its membership, and processes can be added and removed from the process group by the execution of a group membership protocol. A process is permitted to be a member of more than one process group, thereby resulting in intersecting process groups.

The services of a process group can be invoked transparently, with no knowledge of its exact membership or the location of its member processes. Thus, a process in the system can address all of the members of a process group (including its own) as a whole, using a multicast group communication system, such as Totem. A process can send messages to one or more process groups, of which it may or may not be a member. These messages are totally ordered within and across all receiving process groups.

The Totem system provides reliable totally ordered multicasting of messages to processes in process groups. Each message is assigned a unique timestamp, and these timestamps establish the total order of delivery of messages to the application. For messages multicast and delivered within the same configuration of processors, Totem provides these message delivery guarantees despite communication and processor faults, message loss, and network partitioning. The process group layer takes advantage of these services and guarantees of the underlying Totem protocols to provide reliable totally ordered multicasts within and across process groups.

4.2 Object Groups

Analogous to the notion of a process group, an object group is a collection of objects that cooperate to provide some useful service, as shown in Figure 3. This abstraction enables a client object to invoke the services of a server object group transparently, as if it were a single object. The server object group can also return the results to a client object group transparently, as if it were a single object.

An object group may consist of similar or dissimilar objects. In the Eternal system, a replicated object is represented by an object group, the members of which are identical and are the replicas of the object. Both client and server objects can be replicated and thus can be represented as object groups. The reliable totally ordered multicasts of Totem are used to communicate the invocations to, and the responses from, the object group. The replicas of an object receive the same operations in the same order, thereby ensuring consistency of the states of the object replicas. The exact location of the replicas of the object, the degree of replication, and the type of replication (active or passive) is transparent to an object that invokes the services of a replicated object.

5 The Eternal Interceptor

The Eternal Interceptor is a user-level layer between the ORB and the operating system. The principle underlying the design of the Interceptor is that the functionality of an

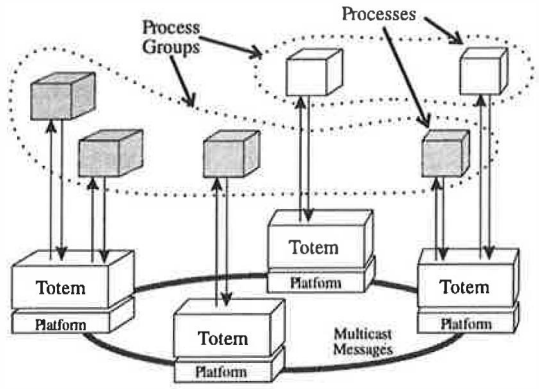


Figure 2: Process groups in Totem.

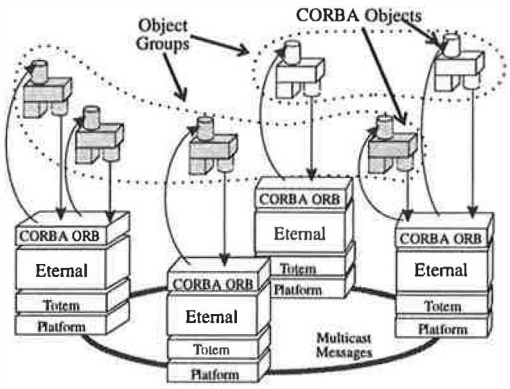


Figure 3: Object groups in Eternal.

operating system can be extended at the user level, without requiring modifications to the kernel or to the standard system libraries. One way of doing this is by intercepting system calls from specified processes before these calls reach the kernel, and then modifying these system calls to implement the desired functionality. The mechanisms are entirely transparent to an application process whose system calls are intercepted.

Such an approach is useful for the development of global file systems [2] and for the testing of kernel extensions. The Eternal system employs the same approach to ''attach'' itself transparently, via the Eternal Interceptor, to all objects that operate over a CORBA 2.0-compliant ORB.

5.1 Intercepting System Calls

The Eternal system can ''attach'' itself to any CORBA object and can ''catch'' a specified set of system calls that are made by the ORB during the object's interactions with the system. To do this, the Interceptor, given the process identifier *pid* assigned by Unix to the object being intercepted, locates and performs a continual trace on the file */proc/pid*, which is part of the */proc* file system of the Unix system.

The system calls to be captured at either the entry to, or the exit from, the system call can be specified *apriori*. In the normal course of events, these system calls would reach the kernel and be executed. However, in Eternal, the tracing facilities provided under the */proc* interface are exploited to enable the specified system calls to be intercepted before they reach the kernel. The arguments, and possibly the return values, of these system calls can be extracted and examined, and the system calls can themselves be modified before they are forwarded to the operating system. Furthermore, all of these mechanisms can be implemented without the intercepted object being aware of their existence.

The obvious advantage of such an approach is that the operation of the Interceptor is transparent to both the CORBA objects and the ORB itself. This functionality can be implemented entirely at the user level, with no modification to the operating system. The application objects and the ORB need not be recompiled to take advantage of the intercepting capability. Once the Interceptor is started, it waits to receive a message from any newly created CORBA object. As a part of its initialization phase, every object supplies its Unix process identifier *pid* to the Eternal system. The Interceptor then monitors */proc/pid* for the entire lifetime of the object.

A typical CORBA object invokes many system calls during its lifetime. These include calls for memory allocation, runtime library access, file operations and network operations. While some of these calls may be local to the machine, any system call that constitutes communication with another object, whether local or remote, must take place using the ORB. The system calls of interest are those that are used by the objects to communicate over IIOP.

All CORBA objects in Eternal use the IIOP interface. The IIOP interface is a simple generic interface to TCP/IP, which makes capture of its calls easy. Since we are only interested in system calls that are IIOP-specific (communication-specific) and not object-specific, the system calls that need to be intercepted are the same for all of the objects operating over the CORBA ORB. Since interception of the calls is transparent to the ORB, any off-the-shelf commercial CORBA ORB, that is capable of communicating over IIOP, can be used unmodified.

Once the system calls of IIOP are intercepted by Eternal, the relevant arguments are extracted from the system calls and passed to the process group layer for communication over Totem. However, the ORB is unaware that its messages are delivered by Totem, since it ''believes'' that it is using only the IIOP interface, the calls of which were originally intended for TCP/IP.

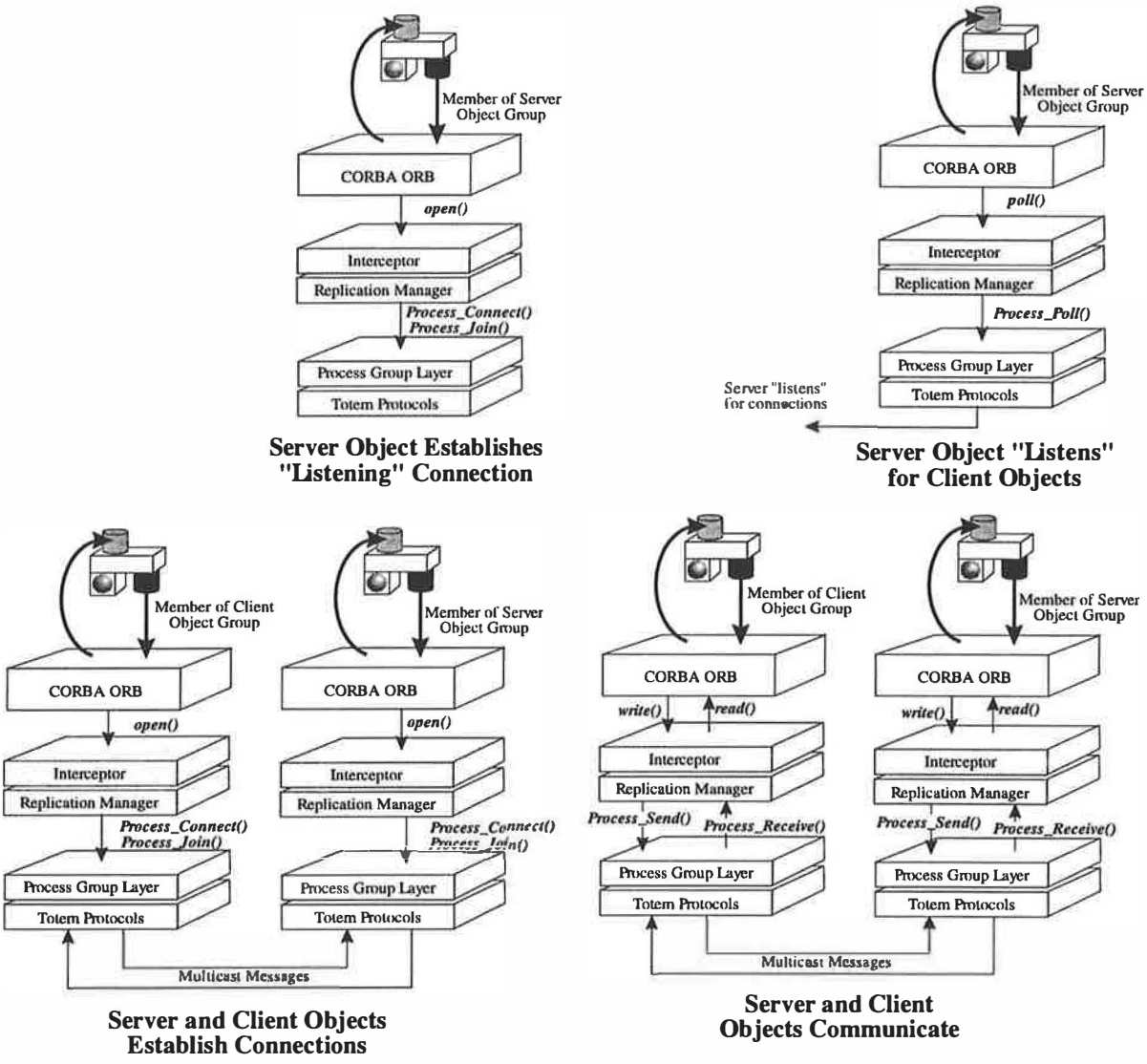


Figure 4: Mapping the IIOP interface onto the process group interface.

5.2 IIOP-Specific System Calls

Each time a server object publishes its identity or each time an object interacts with any other object in the system, the IIOP interface is used. If the ORB's native protocol is IIOP itself, this use is unnecessary.

5.2.1 *open()* System Call

Since the IIOP interface uses TCP/IP, the *open()* system call to TCP/IP is among those intercepted. The file descriptor returned from this call is recorded so that it can be monitored for activity by the system. There are two cases in which an *open()* call may be invoked. In the first case, a connection over TCP/IP is established by a server object that publishes its Interoperable Object Reference (IOR) across the network and ''listens'' for any client object that requires its services. The second case occurs when a client object requests service from a server object and the two objects establish separate connections to TCP/IP in order to communicate.

Thus, each server object has a principal TCP/IP connection, on which it ''listens'' for clients, and establishes additional TCP/IP connections when the client objects desire to communicate with the server object. The additional TCP/IP connections are typically open for the lifetime of the client objects, while the principal ''listening'' TCP/IP connection is open for the lifetime of the server object.

The first *open()* call to TCP/IP, in turn, triggers the Replication Manager, via the Interceptor, to establish a

System Calls of the IIOP Interface	Routines of the Process Group Interface
open(*fd*)	**Process_Connect**(*pgid*), **Process_Join**(*pgid*)
close(*fd*)	**Process_Leave**(*pgid*)
read(*fd*, <buffer to read data into>)	**Process_Receive**(<buffer to receive data>)
write(*fd*, <data>)	**Process_Send**(*pgid*, <data>)
poll(<list of *fd*s >)	**Process_Poll**(<socket to the process group contoller>)

Figure 5: Correspondence between the IIOP system calls and the process group layer routines. Here, *fd* refers to the file descriptor returned from opening */dev/tcp*, and *pgid* refers to the process group identifier. Only the arguments that are relevant to the mapping are shown.

connection with the process group interface of a reliable group communication system, such as Totem, in anticipation of any communication that might follow. Thus, a given object, via the file descriptor associated with this first *open()* call, is associated with a particular connection to the Totem system interface. Subsequent *open()* calls to TCP/IP, which represent client-server communication, are recorded by means of their file descriptors, which are then monitored for any activity. All client-server interactions on these file descriptors can be channelled through the respective connections of the client and the server to Totem.

5.2.2 *poll()* System Call

A server object, on establishing its principal "listening" connection, polls its associated TCP/IP file descriptor, and blocks till it hears from a client object that requires its services. A *poll()* call may also be executed in the middle of a series of client-server interactions, when either object is waiting in anticipation of communication from the object at the other end of the TCP/IP connection. It is also possible for an object to poll several file descriptors simultaneously for activity.

5.2.3 *read()* and *write()* System Calls

Typical communication between objects in the Eternal system consists of a sequence of *read()* and *write()* system calls that operate over IIOP. For each object, these system calls are associated with a file descriptor on which they are invoked. The Interceptor records and monitors all of the active file descriptors associated with an object, and the Replication Manager maps these file descriptors onto the underlying multicast group communication system, in our case Totem. Thus, any system call that uses one of these file descriptors can be mapped to the Totem system interface.

The *read()* and *write()* system calls are associated with receive and send buffers, respectively, that store the information that is received or is to be sent. The contents of these buffers represent the user-level abstractions of the messages that are communicated between client and server objects.

The *read()* and *write()* system calls used by IIOP contain, in the first few bytes, the GIOP header. The IIOP *read()*s and *write()*s are distinguished from other *read()*s and *write()*s by the first four bytes of the data, which represent the **magic** field of the GIOP header, as shown in Figure 6. This field, along with the list of file descriptors associated with the TCP/IP connections, helps in discarding any *read()*s and *write()*s that might not require the IIOP interface and, thus, are not of interest.

5.2.4 *close()* System Call

Since the *open()* system call is intercepted, the *close()* system call for each associated file descriptor must also be intercepted. This call is typically invoked when a client object wishes to close a TCP/IP connection once it has completed communication with a server object. It can also be used to "tear down" the principal connection of a server object, thereby removing it from the CORBA object space.

The *close()* system call, like the *open()* system call, must be handled at both server and client objects. At both client and server objects, Eternal deletes any reference to the file descriptor associated with the connection (that is now being closed). Thus, if the object reuses the same file descriptor for future connections, a new association will be registered.

```
struct MessageHeader {
  char    magic[4];
  Version GIOP_Version;
  boolean byte_order;
  octet   message_type;
  unsigned long message_size;
};
```

Figure 6: Structure of the header of a GIOP message.

5.3 The Process Group Interface

The intercepted *read()*, *write()*, and *poll()* system calls are also mapped onto their corresponding calls in the Totem system interface. It is crucial that the underlying multicast group communication system, in our case Totem, possesses an interface to which the Interceptor and the Replication Manager can map these intercepted system calls.

In order that the services be provided to the application transparently, the group communication system must provide a simple interface to enable the objects that constitute the application to invoke its services. The interface that Totem provides to an application above it is designed to hide the implementation details of the underlying protocols by presenting only a small number of essential primitives that the application needs to use. The interface is intended to be simple and elegant and yet to allow the application to exploit fully the process group mechanisms of Totem.

On each processor, a process group controller manages all of the process groups on that machine. For each process group on that processor, the process group controller maintains information about the member processes (both local and remote) and provides membership services for joining the group, leaving the group and updating the membership. It also maintains a list of the process groups hosted by the machine.

To establish a connection with the process group controller of Totem, the application calls the *Process_Connect()* routine, supplying the identifier of the process group to which the application process wishes to connect. If the process group does not exist, the process connects to a process group of which it is the only member. The routine returns the identifier of the communication socket that connects the process to the process group controller.

To join a process group with which it has established a connection, a process calls the *Process_Join()* routine, supplying the identifier of the process group that it wishes to join, as well as the identifier of the socket between the process and the process group controller. The *Process_Leave()* routine, which takes the same arguments, initiates the removal of a process from the specified process group.

A process can send messages to a process group using the *Process_Send()* routine with the receiving process group identifier and the message to be sent as arguments. A process can receive messages from another process using the *Process_Receive()* routine with the receiving buffer as an argument. The received message is disassembled and the identifier of the sending process is extracted from the message header, along with information about the process groups to which the message is addressed.

The socket between the application process and the process group controller can be polled for any messages pending delivery, using the *Process_Poll()* routine. Routines are also supplied to close the communication socket, once the process disconnects from the process group controller.

```
while Interceptor is running do
 listen for any newly created CORBA objects
 for each CORBA object created do
  obtain process identifier pid and interface name of the object
  obtain object group identifier ogid from the Replication Manager
  specify the system calls (those used by IIOP) to intercept
  while the object is operational do
   wait to intercept the specified system calls when they occur
   case <system call intercepted>
    open() :
      if first open() on /dev/tcp then
       record this as the primary file descriptor for this ogid
       invoke Replication Manager to handle this system call
      endif
      if subsequent open() on /dev/tcp then
       add file descriptor to the list of descriptors for this ogid
       obtain ogid of the object at the other end of the connection
      endif
    close() :
      if server and close() on the primary file descriptor then
       invoke Replication Manager to handle this system call
      endif
      if client and close() on the last open file descriptor then
       invoke Replication Manager to handle this system call
      endif
    poll() :
      if poll() on the previously recorded file descriptor then
       invoke Replication Manager to handle this system call
      endif
    read() :
      if read() on the previously recorded file descriptor then
       invoke Replication Manager to handle this system call
      endif
    write() :
      if write() on the previously recorded file descriptor then
       invoke Replication Manager to handle this system call
      endif
   endcase
   resume the operation of the object
  endwhile
 endfor
endwhile
```

Figure 7: Algorithm executed by the Interceptor.

The calls on the IIOP interface are mapped, through the Interceptor and the Replication Manager, to the process group interface of the Totem system. The implementation of Eternal makes it possible to use any multicast group communication system, as long as it provides the same fault tolerance guarantees and a similar process group interface as Totem. The set of routines that the Totem process group interface uses facilitates the mapping of the IIOP calls onto Totem. The routines can be provided as a library that is used by the Interceptor and the Replication Manager.

```
obtain interface name of object from the Interceptor
look up the table of mappings of interfaces to object groups
if interface name present in the table then
   extract the object's object group identifier ogid
else
   assign a unique object group identifier ogid
   record the interface name and its ogid in the table
endif
communicate the ogid for this interface name to the Interceptor
```

Figure 8: Algorithm executed by the Replication Manager to assign a unique process (object) group identifier for each object.

5.4 Mapping IIOP to Totem

The system calls of the CORBA objects communicating over the IIOP interface are analogous to the routines that the process group interface presents to an application process. In this context, an application process corresponds to a CORBA object in the system, and the process group identifiers of Totem correspond to the object group identifiers of Eternal.

The *open()* system call to TCP/IP corresponds to the *Process_Connect()* routine of the process group interface, since both the call and the routine are involved with the establishment of connections. The assignment of the object (process) group identifier is handled by the Replication Manager, as discussed in Section 6.

The *close()* system call on an open file descriptor corresponds to the *Process_Leave()* routine only if the file descriptor involved is the principal one, since in this case both the call and the routine correspond to the "tearing down" of established connections. If the *close()* system call is invoked on any file descriptor other than the principal one for a server object or the last open file descriptor for a client object, the *close()* call simply causes the Replication Manager to remove any association of the file descriptor with the object group identifier. The *Process_Leave()* routine effectively disconnects the process from the process group controller and is, thus, invoked only when the object is to be destroyed or removed from the object space.

The *read()*, *write()*, and *poll()* system calls find their counterparts in the *Process_Receive()*, *Process_Send()*, and *Process_Poll()* routines of the process group interface. The send and receive buffers of the application objects contain the information that is sent or received over Totem via the process group interface. However, the captured *read()* and *write()* system calls cannot be mapped directly onto the routines of the Totem process group layer since the Interceptor must first associate the file descriptors in the system calls with the process group identifiers in the process group interface routines.

There may be several CORBA objects to which the Interceptor "attaches" itself. Each such object may service multiple requests at the same time, which means multiple connections must be managed for each object. However, for the purposes of replication, an object is associated with only one object (process) group, all the members of which are identical. Thus, each replica of a replicated object is a member of an object (process) group with a unique object (process) group identifier.

The functionality of the Interceptor is implemented using the algorithm shown in Figure 7. The Interceptor does not handle all aspects of the object group mechanisms; it utilizes the services of the Replication Manager for this purpose.

6 The Eternal Replication Manager

6.1 Assignment of Object Group Identifiers

The object groups that are used for replication are handled by the Replication Manager. At creation time, when an object informs the Interceptor of its Unix process identifier, it also conveys the name of its interface. The Interceptor hands this information over to the Replication Manager, which associates a unique object (process) group identifier for each interface name. Since the interface name, rather than any ORB-specific name, is used for this association, objects implementing the same interface, but operating over different ORBs, can be members of the same object group and are treated as replicas from this viewpoint.

The Replication Manager maintains a globally accessible table of the mapping between object (process) group identifiers and interface names. Each time an object is created, if it is the first replica of the object in the system, an entry is created in this table for the object's interface and a unique object (process) group identifier is assigned to it. When further replicas of the object are created and distributed across the system, this table is referenced to ensure that all of the replicas of the object are assigned to the same object group. The object group identifier is assigned or discovered by the Replication Manager, on behalf of the object, using the algorithm shown in Figure 8. In complete implementations of CORBA, the Interface Repository, which stores the interface definitions, can be used to register the object group identifier associated with each interface name.

When a server replica opens its "listening" connection, it discovers its object group identifier and joins its object group. When a client replica wishes to establish a connection to this server object, it must first discover its own object group identifier and join its object group. The client replica then must discover the object group identifier of the object

```
while Replication Manager is running do
 obtain an intercepted system call of the object with the arguments
   case <system call intercepted>
    open() :
      execute Process_Connect() using the object's pgid
      execute Process_Join() using the object's pgid
    close() :
      execute Process_Leave() using the object's pgid
    poll() :
      execute Process_Poll() using the object's pgid
    read() :
      extract the data part of the system call
      execute Process_Receive()
    write() :
      extract the data part of the system call
      obtain the receiver's pgid using the file descriptor in the call
      execute Process_Send() using the receiver's pgid
   endcase
endwhile
```

Figure 9: Algorithm executed by the Replication Manager to communicate messages over the process group layer.

at the other end of the connection (the server object, in this case), and record the association between the connection file descriptor and the server object group identifier.

Once this association is registered, the *read()*, *write()*, and *poll()* calls made by the client object on the file descriptor can be intercepted and subsequently mapped appropriately by the Replication Manager to the known server object (process) group identifier, as shown in Figure 9.

At the server object, the Replication Manager extracts the client's object group identifier from the information packed by Totem into the client object requests that arrive. The Replication Manager then associates this information with the file descriptor of the TCP/IP connection established by the server object to communicate with the client object. Thus, intercepted system calls on the file descriptor at the server object can also be similarly mapped to the appropriate client object group identifier.

6.2 Detection of Duplicate Operations

In the course of their interactions with other objects, the replicas of an object may give rise to duplicate invocations and responses. These must be suppressed at the sender or the receiver since duplicate operations on an object can potentially corrupt its state. The Eternal system accomplishes the detection and suppression of duplicate operations by means of operation identifiers [13], which are assigned by the Replication Manager.

When a replicated object transmits requests or responses, the Replication Manager ensures that the object's own object group identifier is included in the list of object groups that are to receive the request message. This does not imply, however, that the replicas in the object's own object group consider the incoming request message as an operation to be performed. The ''loopback'' mechanism serves only to notify the object's own object group of the transmission of the request.

Thus, every replica of the object that receives messages containing the invocations or responses of another replica in the same object group can suppress its own invocations or responses. The Replication Manager detects these duplicate operations by extracting the operation identifier from the messages that it receives from the process group layer, and then comparing the identifier with those it has already received. If the Replication Manager has already received a message containing this invocation or response, it discards the message, thereby preventing it from reaching the object and corrupting its state.

6.3 Replication Schemes

Eternal is equipped to handle both active and passive replication in a manner that is transparent to the ORB, as well as to each replicated object. Active replication, in which each operation is performed by every replica of the object, requires the detection and suppression of duplicate operations, as well as the use of the object group mechanisms.

Passive replication, in which only a designated primary replica performs each operation, requires additional mechanisms to ensure consistency of the states of the replicas. The Replication Manager performs a state transfer from the primary replica to the secondary replicas at the end of each operation. Thus, after the primary replica completes each operation, the Replication Manager of the primary replica multicasts the primary replica's updated state to the object group containing the primary replica.

7 Conclusion

The Eternal system is a CORBA 2.0-compliant system that enhances CORBA by providing replication, and thus fault tolerance, in a manner that is transparent to the application and to the ORB. The ORB can employ these replication mechanisms without having to undergo any modification to its internal structure.

We are currently implementing the Eternal system using various commercial implementations of CORBA, including the CORBA-compliant Inter-Language Unification (ILU) [4] from the Xerox Palo Alto Research Center. The techniques used in Eternal are completely generic and can interwork with any commercial CORBA implementation that is capable of communication over IIOP.

The replication of objects finds its use not only in achieving fault tolerance, but also in allowing system hardware

and software to be replaced transparently, thereby permitting the evolution of a system with no interruption of service to the application. In addition to the Replication Manager, the Eternal system provides a Resource Manager and an Evolution Manager that handle these challenging issues.

References

[1] D. A. Agarwal, *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara (August 1994).

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser and C. J. Scheiman, "Extending the operating system at the user level: The Ufo global file system," *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA (January 1997), pp. 77-90.

[3] P. Felber, B. Garbinato and R. Guerraoui "Designing a CORBA group communication service," *Proceedings of the IEEE 15th Symposium on Reliable Distributed Systems*, Niagara on the Lake, Canada (October 1996), pp. 150-159.

[4] B. Janssen, D. Severson and M. Spreitzer, ILU 1.8 Reference Manual, Xerox Corporation (May 1995), ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[5] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, John Wiley & Sons Publishers, New York (1997).

[6] C. A. Lingley-Papadopoulos, *The Totem Process Group Membership and Interface*, M.S. Thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara (August 1994).

[7] M. C. Little and S. K. Shrivastava, "Object replication in Arjuna," Broadcast Technical Report 93, Esprit Basic Research Project 6360, University of Newcastle (1994).

[8] S. Maffeis, "Adding group communication and fault-tolerance to CORBA," *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, CA (June 1995), pp. 135-146.

[9] S. Maffeis and D. C. Schmidt, "Constructing reliable distributed systems with CORBA," *IEEE Communications Magazine*, vol. 35, no. 2 (February 1997), pp. 56-60.

[10] G. Minton, "IIOP specification: A closer look," *UNIX Review*, vol. 15, no. 1 (January 1997), pp. 41-50.

[11] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4 (April 1996), pp. 54-63.

[12] Object Management Group, *The Common Object Request Broker: Architecture and Specification* (1995), Revision 2.0.

[13] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Replica consistency of CORBA objects in partitionable distributed systems," *Distributed Systems Engineering*, vol. 4 (September 1997), pp. 1-12.

[14] J. Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons Publishers, New York (1996).

[15] R. M. Soley, *Object Management Architecture Guide*, Object Management Group, OMG Document 92-11-1.

[16] D. C. Sturman and G. Agha, "Extending CORBA to customize fault-tolerance," Technical Report, Department of Computer Science, University of Illinois (1996).

[17] S. Vinoski, "Distributed object computing with CORBA," *C++ Report*, vol. 5, no. 6 (July/August 1993), pp. 32-38.

[18] S. Vinoski, "CORBA: Integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 14, no. 2 (February 1997), 46-55.

Gold Rush: Mobile Transaction Middleware with Java-Object Replication

Maria A. Butrico, Henry Chang, Anthony Cocchi,
Norman H. Cohen, Dennis G. Shea, Stephen E. Smith

IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

{butrico,hychang,tony,ncohen,shea,steve}@watson.ibm.com

Abstract. Gold Rush is middleware supporting the writing of Java applications that reside on an intermittently connected mobile client device and access an enterprise database on a central server. While the client is connected to the central server, objects constructed from database entities can be cached in a persistent store on the client. While the client is disconnected, these entities can be manipulated within transactions that are logged on the client. Upon reconnection, the client application can replay these logged transactions to the server, modifying the database. A replayed transaction is checked for conflicts with other database updates that have occurred since the client obtained the input data for the transaction, and the client is notified when such a conflict arises. Communication between the client and the server is optimized to economize the use of a slow or expensive connection such as a radio link.

Introduction

Continued rapid advances in mobile computing are allowing computing and communication technologies to be applied where they are most effective and productive. For a traveling salesperson, this is typically the customer's place of business. For a health-care worker, it is the point of care.

Disconnecting the client platform from a high-speed network, to provide workers with access to enterprise data when and where they need it, presents many difficulties. One such difficulty is the limited capability of the mobile worker's platform (e.g. processor speed, disk capacity, battery life). Another is the nature of the mobile communications link (e.g. low bandwidth, high cost, frequent disconnections).

Application connectivity requirements for data access vary widely. Some applications require high degrees of connectivity even while mobile. For example, a stock trader requires essentially instantaneous access to the constantly changing prices of stocks while the market is open, and the ability to execute a trade quickly. This requires the trader to pay the high cost of wireless communication, or to be tethered to a phone line while out of the office.

In contrast, a financial planner can perform most tasks while disconnected from the network. The financial planner might begin the day by connecting to a central server and downloading data, such as current market conditions and the portfolios of customers to be visited, from the enterprise financial database to the client device. While visiting with a customer, disconnected from the network, the financial planner might run a Java application on the device to explore various "what-if" scenarios. If it becomes necessary to obtain additional data that was not previously downloaded, the planner could connect to the central server with a brief phone call, download the missing data to the client machine, disconnect again, and resume the use of the program. If the client decides to change the portfolio, the financial planner could execute a local transaction ordering the change. The planner could connect to the central server immediately to replay the local transaction on the central server, or connect at the end of the day to replay all the day's local transactions at once.

In this paper, we describe Gold Rush, middleware that provides lightweight, platform-independent mobile clients with object-oriented, transaction-based access to enterprise information over a weakly connected or primarily disconnected link. Gold Rush includes a client-side persistent store, an object-replication layer to track and minimize data traffic, and an intelligent transaction replay engine. The middleware helps facilitate the development of mobile applications. These are applications that, like the financial planning application,

enable off-line, occasionally-connected workers to execute transactions on enterprise data. Gold Rush enables the financial planner to replicate part of his financial and customer databases, to execute off-line transactions and log them on the disconnected client device, and finally to transmit the logged transactions back to the enterprise's financial and customer databases, checking for conflicting updates.

Gold Rush is not appropriate for every mobile application. Gold Rush is most useful for applications that execute in an environment where client systems are disconnected most of the time. For example, Gold Rush was *not* designed to address the needs of the stock trader who needs continual real-time access to database information. While the device used by the stock trader might in fact be mobile (more precisely, untethered), from the point of view of network connectivity the device is always connected. There are interesting problems that must be solved to keep a wireless communication protocol operational in this environment, but those problems are beyond the scope of the Gold Rush project. In general, applications where the central database changes rapidly, and where the latest version is always needed for a transaction, are not implementable in an occasionally connected environment.

The suitability of Gold Rush for a given application also depends on the application's frequency of conflict. If the application naturally exhibits a low degree of conflict, then it is well suited for the Gold Rush environment, which will allow for conflict resolution. However, if the application typically generates a large amount of conflict, then the application ought to be redesigned, or executed in a connected environment. For example, a set of transactions that always updates a shared counter cannot run in a disconnected environment without conflicting at every transaction replay. A solution in this case is to avoid updating the shared counter at the disconnected client, and to do so later when the transaction is being replayed at the server. That is, the client transaction can be redesigned to specify the amount by which the count should be increased or decreased rather than the value by which the count should be replaced. Given this redesign, the application cannot make use of the actual value of the shared counter. However, if the application cannot be redesigned to avoid using the current value of the shared counter, then the application should connect to commit each transaction.

Off-Line Transaction Requirements

Mobile client applications require access to enterprise data. It is not practical to rely on a constant wireless connection to a server for this access, because radio devices drain batteries quickly and because any-time, any-place wireless links are expensive. Therefore, we replicate enterprise data from the database server on mobile client devices.

The problem is how to write business applications to run seamlessly in both connected and disconnected off-line modes. Will different applications be needed for off-line mode, or can the same application be used with some features disabled? What requirements are imposed by a mobile application, beyond those of a classical client/server connected-mode application? Which data should be replicated, how should it be replicated, and how should the replicated data on the client be synchronized with the data on the main server? How might conflicts arise, and how should they be resolved? To answer these questions, we must examine the characteristics of both business data and off-line business applications.

Off-line business applications are structured into transactions to guarantee atomicity, concurrency control, and durability of the data. A transaction represents a set of read/write operations upon business data. Transactional access enforces integrity constraints: Noncompliant transactions are aborted and accepted transactions are committed into the accumulated state of the database. An off-line transaction commits data into a client's local store while the client is disconnected. The commit is replayed to the master database when the client is reconnected. This approach of a *lazy* commit, consisting of two stages, is necessitated by the occasionally connected nature of mobile applications. In an analytical paper [GH96], Gray et al. compare several transaction propagation strategies and conclude that the lazy-commit approach is the one that scales well and fits into mobile environments.

Conflicts may arise during reintegration with the main database. For example, if multiple clients change the same field while they are disconnected, the conflict can only be detected during reintegration. The previously committed data of the disconnected clients might then be reconciled using a predetermined formula based on the context of the transaction or the data itself.

The flow of the off-line transaction model of application development can be summarized as follows:

1. ***Check out:*** partial replication of business data from the business server together with its integrity constraints
2. ***Access:*** off-line transactional access with all the read/write information logged
3. ***Check in:*** reintegration of off-line transactional data with main database
4. ***Conflict handling:*** detection and resolution of conflicts with predefined formulas or repair utilities

Transactions provide off-line, occasionally connected business applications with a design model for data integrity and conflict prevention. Without such a model, we would not be able to determine the scope of conflict and to make proper repairs to reintegrate off-line data with the master database. Our model of off-line transactions is similar to the transaction model of network-partitioned databases, but less stringent on the client side, because the mobile client is a single-user, one-application-at-a-time system, and playing a second class role relative to the master database.

Related Approaches

Existing Java remote-database-access products based on the JDBC API [Ja97] are designed for permanently connected clients. These include IBM's VisualAge for Java [IB97] and Symantec's dbANYWHERE [Sy97]. In contrast, Gold Rush supports an occasionally connected client. JDBC provides access to data in terms of relational-database tuples. In contrast, Gold Rush supports manipulation of Java objects.

Several methods have been proposed to allow mobile workers to access information from central data bases. The methods are as varied as the type of information: files, relational data bases, web pages, etc. For mobile file access, the Coda remote file system of CMU pioneered the notion of disconnected operations [KI92], based on file-level transactions isolating groups of changes to files by an application [LS94]. For mobile access to documents, Lotus Notes [Lo96] handles two-way data replication, allowing document-level and field-level propagation, but without grouping changes into transactions. Several approaches have been suggested for access to database information. Of greatest interest to us are those methods which not only make the information available for reading, but also allow changes to be written.

An alternative approach is to download a portion of the central database to a private database on the mobile client. The smaller database is accessible through traditional interfaces, such as ODBC or JDBC, residing on the client. The application makes all modifications to the smaller database. When the mobile client is able to communicate again with the central database, the changes made to both databases are reconciled. The reconciliation is carried out by software usually called a *replicator*.

If the replicator detects a conflict during reconciliation, it acts according to its configuration. Typically, replicators can be instructed to carry out some default action in case of conflict—for example, merging the changes if possible, or discarding the tuple with the older timestamp—and also allow for custom-programmed actions.

Replicators are often tightly coupled with the implementations of the database systems both in the mobile client and the centralized system. In addition to a complete replicator, that is, one that can incorporate changes from both sides, this approach requires the availability of a suitable small server that can host the smaller database on the mobile client.

This design, and the suitability of the replication and reconciliation mechanism in mobile or other environments, have been studied in depth. [GH96] points outs the instability of some replication methods and proposes algorithms that alleviate this problem; [Fr96] addresses scalability and availability issues; [RZ96] discusses the effect on transactions on disconnected operation, and proposes a transaction management model; [YT96] proposes optimistic concurrency control, and addresses migration and replication methods; [Pi96] presents a method for replication in the presence of challenging connections; [ZF96] proposes another replication method and analyzes its performance; [Wo95] evaluates yet another strategy using an application for travel agents; [YW94] describes an algorithm for dynamic allocation of replicas; [AN93] discusses replication organization, and reconciliation methods. Finally, major database vendors offer database-access products for mobile workers based on this approach ([Sy96], [Or95], [IB95]).

A replicated database burdens the mobile client with a database server and with logic to access the data stored on this server. Recently, the increasing popularity of Internet and intranet applications has made light-weight clients desirable. Rather than placing a database server on the mobile client, a three-tier architecture with *mobile transaction middleware* gives the client access to server data without tying the client to a specific database implementation. Three-tier systems move the interface between the application and the database to a central server. The Tactica Corporation has a commercially available product, Caprera, which supports off-line *long-lived transactions* and three-tiered access to databases [La96].

Our Approach

In a mobile database application, *mobile transaction middleware* provides mobile connectivity and mobile data management. The mobile middleware provides support for:

- a wire-efficient access protocol
- object caching and replication
- logging of deferred transactions
- a server-side object server to reduce the frequency and duration of slow-link connections

It is not sufficient simply to extend database query capability to the mobile client. There must be services to manage the data for mobile use.

Gold Rush mobile data management is based on Java objects. Java has attracted wide interest because it facilitates cross-platform deployment. Furthermore, the Java Remote Method Invocation (RMI) API [Su96] supports remote method-call and object-shipping paradigms, which are useful for both connected and disconnected operations. Java technology is very well suited for mobile database-access applications.

An objective of Gold Rush is to make enterprise data available to Java applications. Enterprise data is most likely to be found in relational databases, VSAM data sets, or IMS databases; a very small portion of such data is in object databases. Typically, a one-time conversion of these relational data bases into object data bases is not possible because of other existing applications that regenerate and alter the data stored in them. Our current prototype supports mappings between relational data base tuples and Java objects.

In a connected environment, one can use a remote method call to access an enterprise database or to download an object for temporary caching. In an occasionally connected environment, one must first download Java classes and data to the Java-enabled client. Java applications or trusted applets can then support disconnected operations through locally persistent objects. Application code can manipulate local and remote objects uniformly, through the same object interface.

The Gold Rush three-tier architecture consists of a Java client, an intermediate mobile object server, and a back-end data store (see Figure 1). The mobile middleware resides partly on the client and partly on the intermediate server. The middleware presents the client application with the same transaction API regardless of connection mode, except that the database cannot be queried in disconnected mode. Thus, a method call that would obtain a service directly from the server in connected mode invokes middleware that transparently performs that service locally in disconnected mode, using locally available resources.

This is not to say that the application programmer is oblivious to the mobile nature of the application. The parts of the application that are specifically mobile and must be exposed to the programming interface include the handling of the modes of connectivity, prefetching and downloading objects, controlling the replay of transactions, and resolving conflicts during reconnection.

The Gold Rush middleware has the following basic components (see Figure 2):

- ***Database objects:*** A *database object* is a Java object that represents an instance of a database entity.
- ***Object caching:*** To support disconnected transactions on data base objects, we provide a persistent local object store on the client.
- ***Transactions with optimistic concurrency control:*** The client works primarily off-line on data objects in the mobile device. Transactions are logged on the client side and replayed to the server when connection is established. An object read by a transaction may be either locked with an optimistic read lock or left unlocked. An object written by a transaction is locked with an optimistic write lock.

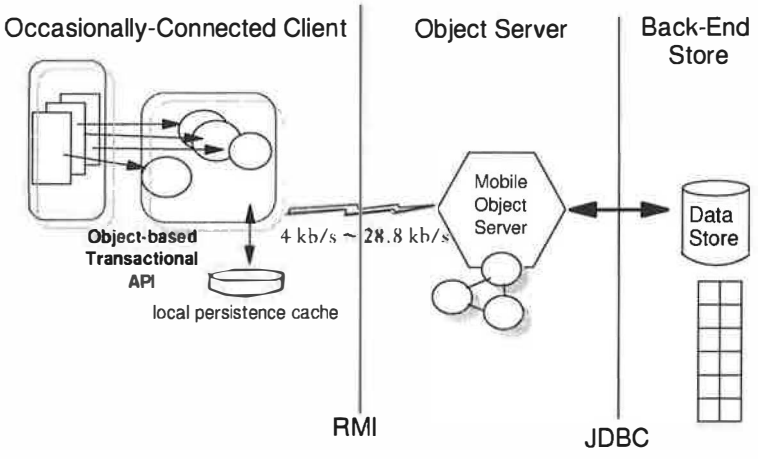


Figure 1. Our three-tiered architecture

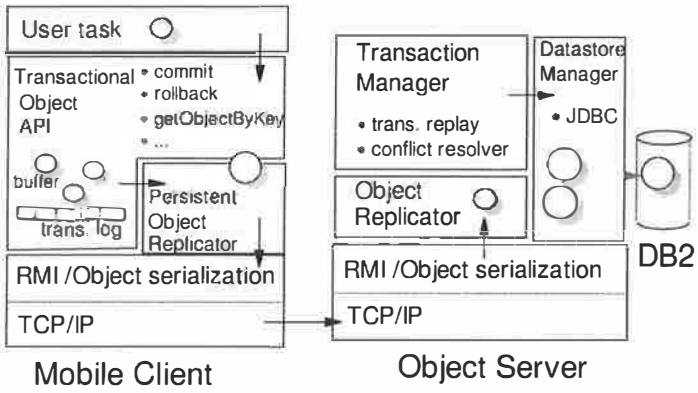


Figure 2. Mobile middleware components

- ***Communication:*** Database objects are transferred between client and server using RMI and Sun's object serialization mechanism. To optimize this communication, we keep track of objects known to be present both at the client and at the server and transmit only the differences between the object version to be transmitted and the version, if any, known to be stored remotely.

The following sections describe the correspondence between database objects and relational database entities, the persistent client object store, the off-line transaction model, and optimization of communication.

Correspondence Between Relational Data Bases and Java Objects

A database object is a Java object that corresponds to a row in a relational database table. Each such object belongs to a subclass of a Java class named `Entity`. Each such subclass corresponds to a table of the relational database.

In a relational database, tuples in different tables are related through primary and foreign keys. We distinguish among 1-1, 1-*n*, and *m*-*n* relationships. (1-1 relationships are special cases of 1-*n* relationships.) If a 1-*n* relationship exists between two object classes, then the table corresponding to one object class must have a foreign key into the table corresponding to the other class. If there is an *m*-*n* relationship between two object classes, then there must be a third table with foreign keys into both of the tables corresponding to the related classes.

Our system provides methods to retrieve database objects that satisfy queries, for example, a query on foreign keys, when the client is connected to the server. To allow retrieval of these collections when the client is disconnected, we provide methods to associate names with collections. These named collections are persistent at the client.

An application using our system would include a layer to insulate the details of relational database storage, such as foreign keys, from the manipulation of the objects themselves. Such a layer would provide a subclass of `Entity` for each kind of database object, defining the object's properties and its methods. This class would also supply methods for navigation among objects, using the Gold Rush query facility, and associate unique names with collections to allow the association between objects to persist. Finally, the application would supply a *data manager* class for each subclass of `Entity`, establishing the correspondence between object properties and database fields and implementing the database retrieval and store function.

We have written a tool that allows the application developer to map relational data to object classes. This tool generates the code that defines the classes of database objects; the code needed to instantiate the objects from tuples in the relational database and write object instances into the relational database; and the code that navigates between related object instances.

The mapping tool is a Lotus Notes application. We chose Lotus Notes because it offers flexible storage to represent the association between objects' attributes and fields in tuples, a fast way to develop the user interface through which the programmer establishes this association, and a sufficiently powerful programming language to support the code generation.

It is possible to map a subset of one tuple to a subset of an object. In practice, however, an entire tuple is typically mapped to an entire object. The mapping tool does not support the mapping of an object

to multiple tuples, whether from the same table or from multiple tables.

The tool allows the relationships between tuples to be reflected in relationships between object instances, through generated code which allows the application to retrieve related object instances, or establish relations among object instances. Code is generated for `setXXX` and `getXXX` methods (where `XXX` represents an attribute of a database object) in both classes corresponding to a relationship. A `getXXX` method retrieves the associated object instances (one instance in the case of 1-1 relationships); a `setXXX` method establishes such relations.

For each subclass of `Entity`, the tool generates a corresponding JDBC-based data-manager class that provides SQL statements for retrieval, insertion, and update. Each data-manager class provides a method that retrieves data from a particular database table based on an SQL query and returns a Java collection object containing one Java database object for each row in the relational database that satisfied the query. There is also a method to retrieve an object given its unique object ID. The class manager's insertion method takes a Java database object and inserts a new row into the database, checking that uniqueness constraints are not violated. The data manager's update method takes a Java object and replaces one row in the relational database with the data found in the object, checking for update conflicts.

Persistent Client Store

We use Sun's object serialization as the principle means of generating a persistent form of object. To further control the persistent state, and to improve efficiency, we have implemented the serialization methods `readObject` and `writeObject` selectively on certain complex internal objects.

To cache persistent data on the mobile client device, we create a small single-user persistent store. This object store provides object lookup, store, update, and retrieval functions for the objects used by applications in off-line transactions. The main operations of the store are:

- ***Caching:*** Data is cached into the client's persistent store during connected transaction processing. The latest version of an object is saved in the store in preparation for subsequent disconnected operations. (The communication optimizations to be discussed later prevent the redundant transmission of an object already residing in the client's store.)
- ***Retrieval:*** The application code residing on the client reads objects from the persistent store while the client is disconnected. An application may retrieve an object by calling a method that retrieves the latest version of an object with a given ID, or by calling a method that returns a collection containing all the locally stored objects of a particular class.
- ***Committal:*** All mutated objects in the committed transaction are stored and updated. A transaction record is created and the information is stored in the transaction log for future replay.
- ***Replay:*** The client retrieves the transaction log and replays it to the master database, guided by the dependency relationships among the transactions. After replay, the persistent store cleans up itself by removing stale and old versions of objects as well as old transaction information.

Since the client store is a single-user, one-application-at-a-time persistent store, we choose a file-based design with careful write sequencing to guarantee that the on-disk data is always in a consistent state. The store consists of the following kinds of files:

- ***Class files:*** Objects of the same class are stored in a single file. This file contains all versions of all objects of a given class resident on the client. A new version of an object is written when a transaction modifying that object is committed.
- ***Class index files:*** A separate index file is created for each class. The index provides fast look up of object by object ID and transaction ID.
- ***Transaction files:*** A file is created for each transaction when that transaction is committed. This file identifies the exact version of each object involved in the transaction and also identifies the other transactions upon which this transaction depends.
- ***Transaction log:*** There is one file containing a record of each transaction in the system. The log entry for a given transaction includes the name and *state* of the transaction as well as information concerning transactions on which the given transaction depends. The major states of a transaction are *locally committed*, *remotely committed*, and *aborted*.

Off-Line Transactional Semantics and Disconnected Transactions

When transactions are run while the client is connected, locks are held in the database and the transaction runs in the traditional way. We will not describe the connected mode of operation in this paper. While the client is disconnected from the server, locks are not held in the database and the system runs in a "lazy" mode similar to that described in [GH96]. Multiple transactions can be run against objects resident in the

client's local store. When the client reconnects, the transactions are *replayed* on the server.

We perform disconnected transactions using the latest version of each object resident in the client's local store, and save the results in the transaction log. In addition, the changed version of each object modified by the transaction is saved. When the client reconnects, the transaction log is replayed to the server and the final commit to the database is attempted. The initial execution is called a *local commit* and the replay is called a *remote commit*. Locks are granted to the client optimistically before disconnection. These locks are used to detect conflicts during remote commit.

To support conflict detection, each object has an object ID and a timestamp, which are stored in the database. The object ID is generated when the object is created. It is unique, is not modifiable, and is a key of the database table. The object timestamp is unique and is generated locally on the client at the time of commit. It consists of a unique user number concatenated with a local clock reading and a counter to distinguish among objects created during a single tick of the clock.

Gold Rush provides read locks and write locks with the usual semantics (shared read and exclusive write) [Da90]. These locks are not checked during disconnected execution, since transactions proceed in strict serial order on the client, but they are checked during transaction replay (remote commit). It is also possible to read an object without locking it, and without checking for currency when the transaction is replayed.

Reading without locking reduces the amount of lock contention. If a transaction reads an object without locking it, the transaction can commit successfully even if the version of the object read by the transaction is obsolete at commit time. Reading without locking is useful when it is known that the attributes actually used by the application in a read object (for example, the name and serial number in an `Employee` object) are unlikely to change even though other attributes in the object (for example, the employee's accrued vacation time) may change. Reading without locking is also useful when an approximate answer is sufficient to satisfy application requirements.

A transaction is started on the client when the application calls a `beginTransaction` method. After that other methods can be called to register particular objects with the current transaction. The application can use and modify any registered objects as it chooses and eventually call a `commit` method, closing the current transaction. When `commit` is called in disconnected mode, each modified object is written to the proper class store. A transaction file is created with references to each of the locked objects in the transaction. The transaction log is extended to include this new transaction file. At this point the transaction's state is locally committed.

When the client eventually reconnects to the server, it serially reads all locally committed transactions and replays the transactions to the server. Each of these transactions creates an equivalent transaction on the server. The set of locked read objects and the set of write objects are checked for conflicts and the database updated if no conflicts are detected. If no conflicts are detected, the transaction succeeds, and is marked *remotely committed*.

To perform conflict detection the system tracks the following information:

- On the server, each database tuple includes the object ID and the timestamp of the last update. This timestamp is called the *last-modified time*.
- On the client, each modified object includes two timestamps. One is the *last-modified time* and the other is the *local-commit time*.

When an object is first read by a transaction, the *last-modified* timestamp is set to the timestamp of the last local commit that modified the object, or initially to the database timestamp. When the transaction commits, the *local-commit* timestamp is set and the object is written to disk. When the transaction is replayed during remote commit, the object's *last-modified* timestamp is compared to the database tuple's timestamp. If they are the same and the object has been modified, the object will replace the database tuple and the *last-modified* timestamp in the database will be changed to the client object's *local-commit* timestamp. If the object has not been modified, the replay proceeds to the next object in the transaction. If the *last-modified* timestamps are different for any object in the transaction, the transaction is rejected. (No global clocks are required, because timestamps are compared only for equality, not order, and each timestamp includes a user number unique to its client.)

Reducing Data Traffic Between Client and Server

When a mobile client connects to the server, the connection may be over a slow and expensive link such as a cellular phone connection. Therefore, it is important to minimize the amount of data exchanged between the client and the server, even at the cost of additional computation and additional storage requirements.

We reduce traffic between the client and server by maintaining mirrored directories of objects known to be stored on both the client and the server. There is one such directory on each client machine and one mirrored directory per client on the server machine. Each direc-

tory entry contains an object ID and a reference to a local copy of the corresponding object.

Before an object is transmitted remotely, we check whether its object ID is in the local copy of the directory. If not, we transmit the entire object and—since the object is now stored on both the client and the server—add a corresponding directory entry to each copy of the directory. If the object ID is already in the local directory, we compare the timestamp of the object referenced in the directory with the timestamp of the object to be transmitted. If the timestamps are the same, we transmit only the object ID. If the timestamps are different, we transmit both the object ID and a succinct representation of the differences between the version of the object to be transmitted and the version referenced by the directory entry.

We rely on RMI for the actual transport of data between client and server. Entities are transmitted from the server to the client only as the function result of an RMI call by the client asking for an object with a particular object ID, or as elements of the function result of an RMI call by the client asking for a vector of objects satisfying a particular SQL query. Entities are transmitted from the client to the server only as elements of a parameter of an RMI call asking the server to commit a particular transaction.

We do not tamper with the internal mechanisms of RMI to take advantage of our mirrored directories. Rather, we use an abstract class `RemoteEntity` in place of the class `Entity` in the parameters and function results of the RMI call. This abstract class has three subclasses providing concrete implementations:

- `FullRemoteEntity`. This class carries all the information contained in an entity.
- `OidOnlyRemoteEntity`. An object of this class contains only the object ID of an entity.
- `DeltaRemoteEntity`. An object of this class contains only the object ID of a base entity and a succinct description of the changes that must be applied to the base entity to obtain the desired target entity.

The remote method for committing a transaction is called through an interface that accepts vectors of entities participating in the transaction, constructs vectors in which each entity is replaced a remote entity of the appropriate form (based on whether that entity is present in the mirrored directories), and passes these vectors to the remote method. The server converts these vectors back to their original form using its copy of the mirrored directory and performs the commit operation. Similarly, when the client calls a remote method to obtain a particular entity or vector of entities, it does so through an interface that will convert the remote entities returned by the RMI call into ordinary entities, using the client copy of the mirrored directory. The remote method itself, executed at the server, first performs the necessary database operations to construct the result entity or result vector, then constructs a corresponding remote entity or vector of remote entities based on its copy of the mirrored directory and returns that object as the result of the RMI call. (See Figure 3.)

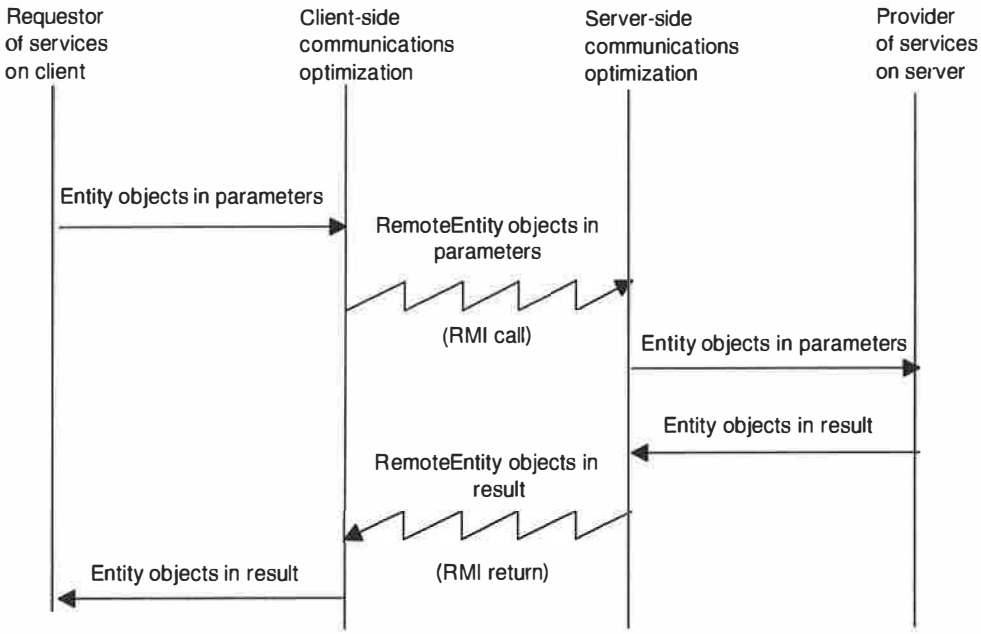


Figure 3. Protocol for optimized RMI calls.

Because we control the amount of data passed in remote calls rather than the mechanisms by which remote calls transmit their data, the fact that we are using RMI is incidental to our approach. The same approach could be used if we were to replace RMI calls with IIOP/CORBA calls. Indeed, by concentrating the bulk of our remote calls in the part of Gold Rush responsible for traffic reduction, we have encapsulated our decision to use RMI. Very little of our code would have to change if we were to decide at some point to use IIOP instead.

The Role of the Application Code

A system that uses Gold Rush contains application code in both the client and the server. In the server, the application code implements the interface between Gold Rush and the persistent repository, typically a relational database. For this interface the application supplies definitions of the objects and methods for:

- querying the attributes of a single object;
- setting the attributes of a single object; and
- retrieving of a set of objects, all of the same class, satisfying a query clause.

In addition, the application must supply methods to resolve conflicts. We also envision that the application code at the server could run more complex requests, perhaps activating an agent to execute autonomously on behalf of the client, and perhaps yielding a result consisting of objects of various classes.

At the client, Gold Rush provides methods for transaction start, commit and rollback, object creation, and lock upgrade. There are also methods for a connected client to create collections of objects of a given class satisfying a given query. Finally, there are primitives for associating names with such collections and making the collections persistent at the client. The application must supply methods to navigate between related objects.

Future Directions

We have implemented a prototype version of Gold Rush. Gold Rush is now being considered for integration into a large business-object framework. Its interfaces were designed to facilitate this integration. Issues that we did not address in the prototype but are important in a production system, such as data security, would be addressed during this integration.

We are aware of a number of ways in which we can improve the performance and flexibility of Gold Rush. We expect to incorporate these improvements in our future work.

Currently, the client is notified when a conflict is detected during an attempt to perform a remote commit. However, any actions to recover from the conflict, for example by merging updates or retrying a transaction with fresh data, must be programmed explicitly. Future versions of Gold Rush will include a framework for specifying conflict-resolution strategies. It will be possible to specify strategies both on a class-by-class basis and on a scenario-by-scenario basis.

Our current mapping between relational data bases and objects entails the addition of columns to the relational database tables to hold object ID and time-stamp data for use in conflict detection. This is inconvenient and sometimes unacceptable when dealing with legacy data bases. We are formulating techniques that will allow access to legacy data bases without changing the format of those data bases.

The current system requires that all data be preloaded before disconnection. This requires careful planning by the user to avoid being stranded without the data needed to complete one's work. We are investigating techniques to dynamically connect to the server and fetch missing data.

Our prototype presumes that the client middleware is invoked by only one application at a time, and that all of the application's invocations of middleware methods are performed by a single thread. Thus there is no concurrency control in the local-commit logic. This is not an inherent limitation in our approach, merely a simplifying assumption made for the prototype version. We plan to rewrite the client middleware to make it thread-safe, so that the mobile worker can run several applications at a time and so that a client application programmer can take advantage of Java threads.

Our current mobile object server utilizes one active remote-object-server object for every client regardless of whether the client is connected or disconnected. This naive approach supports high concurrency but requires large number of thread resources, potentially over a long period of time with huge number of clients. We plan to investigate simple activation approaches such as the one described by Wollrath [WW95] to reuse remote-object-server objects if possible and to activate and deactivate persistent remote objects with low overhead.

Several of our strategies for reducing traffic over a slow or expensive link entail a large amount of computation. Over a sufficiently slow link, the time saved by reducing traffic more than makes up for the time expended to perform the computation. However, on occasions when the client is connected to the server over a fast and inexpensive link (as when a mobile user returns to the office and connects the client machine directly to a LAN), the time saved by reducing traffic is negligible, and the computational cost is no longer worthwhile. Therefore, we plan to provide controls for disabling our computationally intensive traffic-

reduction strategies. Ultimately, it may be possible to monitor the client-server connection and switch modes automatically based on the speed and cost of the connection and the user's current level of urgency (measured as the amount of extra money the user is willing to pay to speed up transmission).

In our present architecture, all application-specific algorithms (except for the formulaic methods that our tools generate to translate between object-oriented and relational data bases) reside on the client. These algorithms communicate with the server through simple-minded requests to fetch an object with a given ID, to fetch a collection of objects satisfying a given query, or to commit a transaction remotely. Some algorithms (executable only in connected mode) may involve an extended dialogue between the client and server, in which the client requests some data and, based on the contents of that data, issues further requests. In such cases, traffic between the client and server could be substantially reduced by allowing the client to issue high-level application-defined requests to the server. These requests would invoke application-specific algorithms at the server and deliver results to the client. A server-based algorithm might entail a long series of database queries and updates, but these would all be performed locally on the server. Only the initial high-level request and the final result would have to be communicated, producing substantial savings when the connection is over a slow or expensive link. The server-based algorithm could even be executed autonomously by an agent acting on behalf of a disconnected client. The client would retrieve the result of the autonomous computation upon reconnection.

Conclusion

Mobile applications in Java can easily be ported to other platforms, and can exploit Java's strong support for distributed applications. Gold Rush allows mobile client code written in Java to access data stored in enterprise relational data bases. These applications deal with Java objects corresponding to rows of relational database tables, belonging to classes that correspond to tables. Attributes of these objects reflect the 1-1, 1-*n*, and *m*-*n* relationships among relational-database entities. We have developed tools that automatically generate the required Java classes and translate between the object and relational views of the data.

Unlike other systems allowing relational-database entities to be manipulated as Java objects, Gold Rush allows users to cache objects off-line on the mobile client and then disconnect, obviating the need for a continual, potentially expensive, link to a central server. A client also has the option of running in disconnected mode when a slow link is available, to avoid communication delays. Unlike systems that replicate a subset of a relational database on the client, our architecture confines all manipulation of relational data bases to central servers. The client deals purely with Java objects and can remain lightweight, using the same object interface for both connected and disconnected transactions.

To guarantee atomicity of updates and the integrity of both the central database and the data stores on individual clients, we group updates into transactions. While the client is connected, transactions can be run directly on the server. While the client is disconnected, transactions are constructed and saved locally on the client and replayed to the server upon reconnection. Objects participating in transactions can be locked optimistically, which allows other clients, or back-office applications, to use the same data, but makes it necessary to check for conflicts when client transactions are replayed to the server. In case of conflict, the central database is not updated and the client is notified of the failure. A conflict-resolution mechanism currently under design will allow the client to take appropriate actions to recover from the rejection of a transaction that had been tentatively committed.

In addition to reducing the need for communication between client and server to the initial caching of objects and the replaying of transactions, we streamline those communications that are necessary. By reducing the amount of data that must be transmitted, we make the use of mobile clients more economical and practical.

Acknowledgement

We thank Professor I-Chen Wu of the National Chao-Tung University at Taiwan for his help with the persistent client store.

References

[AN93] Adly, N., Nagi, M., and Bacon, J. A hierarchical asynchronous replication protocol for large scale systems. Proceedings, 1993 IEEE Workshop on Advances in Parallel and Distributed Systems, Princeton, New Jersey, October 6, 1993. IEEE Computer Society Press, Los Alamitos, California, 1993, 152-157

[Da90] Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, Reading, Massachusetts, 1990

[Fr96] Froemming, G. Moving forward with replication. *DBMS* **9**, No. 4 (April 1996), 83-84+

[GH96] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The dangers of replication and a solution. Proceedings, 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 4-6, 1996. *SIGMOD Record* **25**, No. 2 (June 1996), 173-182

[IB95] IBM Corporation. *An Introduction to DataPropagator Relational, Release 2*, 1995

[IB97] IBM Corporation. VisualAge family web development directions, 2nd ed.. `http://www.software.ibm.com/ad/vajava/wp.htm`, March 1996

[Ja97] JavaSoft. The JDBC(tm) database access API. `http://splash.javasoft.com/jdbc/`, February 4, 1997

[KI92] Kistler, J.J., and Satyanarayanan, M. Disconnected operation in the Coda file system, *ACM Transactions on Computer Systems* **10**, No. 1 (February 1992)

[La96] Lardear, J. Taking transactions off-line: Tactita's Caprera transaction monitor demonstrates emerging transaction-processing model. *MIDRANGE System* **9**, No. 12 (August 1996), 18

[Lo96] Lotus Notes Release 4.5: A Developer's Handbook, IBM Corporation

[LS94] Lu, Q., and Satyanaranyanan, M. Isolation-only transactions for mobile computing. *ACM Operating Systems Review* **28**, No. 2 (April 1994)

[Or95] Oracle Corporation. Oracle 7 distributed database technology and symmetric replication, April 1995

[Pi96] Pitoura, E. A replication schema to support weak connectivity in mobile information systems. Database and Expert Systems Applications: 7th International Conference, DEXA '96 Proceedings, Zurich, Switzerland, September 9-13 1996. Springer-Verlag, Berlin, 1966, 510-520

[RZ96] Rasheed, A., and Zaslavsky, A. Ensuring database availability in dynamically changing mobile computing environment. Proceedings, ADC'96: Seventh Australasian Database Conference, Melbourne, Australia, January 29-30, 1996. Australian Computer Science Communications **18**, No. 2 (1996), 100-108

[Su96] Sun Microsystems. RMI—remote method invocation. `http://java.sun.com/products/JDK/1.1/docs/guide/rmi/`, 1996

[Sy96] Sybase Corporation. SQL Remote: replication anywhere, 1996

[Sy97] Symantec Corporation. Evaluating network database architecture. `http://www.symantec.com/dba/wp_evalnda.html`, January 28, 1997

[Wo95] Wolfson, O. Mobile computing in a reservation application. Information and Communication Technologies in Tourism: Proceedings of the International Conference, Innsbruck, Austria, January 18-20, 1995. Springer-Verlag, Vienna, 1995, 43-45

[WW95] Wollrath, Ann, Wyant, G., and Waldo , J. Simple activation for distributed objects. Conference on Object-Oriented Technologies, June 26-29, 1995,1-11

[YT96] Yoshida, T., and Takizawa, M. Model of mobile objects. Database and Expert Systems Applications: 7th International Conference, DEXA '96 Proceedings, Zurich, Switzerland, September 9-13 1996. Springer-Verlag, Berlin, 1966, 623-632

[YW94] Yixiu, Huang, and Wolfson, O. Object allocation in distributed databases and mobile computers. Proceedings of the 1994 IEEE 10th International Conference on Data Engineering, Houston, Texas, February 14-18, 1994. IEEE Computer Society Press, Los Alamitos, California, 1994, 20-29

[ZF96] Zaslavsky, A., Faiz, M., Srinivasan, B., Rasheed, A., and Lai, S. Primary copy method and its modifications for database replication in distributed mobile computing environment. Proceedings, 15th Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, Ontario, Canada, October 23-25, 1996. IEEE Computer Society Press, Los Alamitos, California, 1996, 178-187

Metis: A Thin-Client Application Framework

Deborra J. Zukowski
Apratim Purakayastha
Ajay Mohindra
Murthy Devarakonda

IBM Thomas J. Watson Research Center,
P. O. Box 704, Yorktown Hts, NY 10598.

Abstract This paper introduces a thin-client programming model and then presents an object-oriented framework for developing applications using the model. The programming model and the framework have evolved from interactions with developers and users of commercial applications. The key aspects of the thin-client programming model are that the client downloads application front ends from the network; that these applications rely only on services found on network servers; that the services are bound as late as possible; and that the applications interact with each other within the confines of a workspace. We implemented the framework using Java Beans and JDK 1.1, and developed several sample applications using the framework.

1 Introduction

Fueled by $Java^{TM}$ and other Internet technologies, new re-engineering efforts are underway to develop commercial applications using a thin-client programming model. In a thin-client programming model, the software client would be substantially *thinner* in that it contains only the graphical user interface (GUI) and a small amount of essential application logic. Most of the application logic runs as *services* on various servers throughout the network. The client software is written using Java so that it can run on any client hardware. The thin-client model is distinct from its hardware counterpart, known in the industry as the Network Computer. However, the thin-client programming model can be the force that makes Network Computers widely deployed.

An application development paradigm becomes popular if appropriate tools are available that enable developers to leverage its benefits easily. While the Java programming language[1], Java Development Kit (JDK) [2], Java component technology [3], and remote access mechanisms [4, 5] enable platform-independent programming, they are only a set of building blocks. Previous work in object-oriented systems suggests that frameworks [6] can be a promising way of achieving widespread use and reuse of software architecture. Therefore, there is a need for a thin-client application framework that is capable of bringing together all parts of an application (the front-ends running on the client and the services available on network servers) and supporting the whole with system services. Lacking such a framework, developers may find it difficult to bootstrap themselves into the new paradigm, and they might resort to an older and less portable methodology such as the Microsoft Windows environment.

Metis, the thin-client application framework presented in this paper, is a related, inter-operable set of objects that enable robust application development in the thin-client paradigm. The goal of Metis is to create a fully server-managed environment for an application, as opposed to the traditional client-server approach. Towards this end, the framework advocates and supports a thin-client programming model where an application consists of application front ends (AFEs) and a collection of backend application-specific services. AFEs rely solely on application-specific services and system services provided by one or more network servers. Thus, AFEs do not depend on local operating system functions. AFEs request services in an abstract manner without specifying the physical location of a service provider. That is, a requested service can be any one of the appropriate service instances available in the network. AFEs

bind, on demand, to these network services. The late binding of services allows server manageability, flexibility, and fault-tolerance.

Metis provides Java classes on the client side for locating and binding to a service instance and for switching to an alternate service instance in case of a failure. In addition, Metis provides a workspace-based client environment suggested by a common commercial application characteristic: interacting sub-applications. The Metis workspace hosts and manages a set of sub-applications; each sub-application is in the form of an AFE. The workspace manager provides visual tools to customize the workspace by adding or deleting AFEs. Workspace configuration information is stored on a server.

In the current implementation, the Metis workspace provides the following object instances for use by the AFEs, and the list may grow as additional objects of common applicability are identified:

- Service location and binding object;
- User authentication object;
- Controller objects for accessing and managing system services such as printing and data storage.

On the server side, the Metis framework depends on support services including an authorization service that ensures controlled access to the system, a code service that maintains a secure repository of trusted AFEs, and a directory service that presents a searchable access to services. These support services must be fault-tolerant and scalable besides using industry standard protocols. Therefore, Metis uses a directory service supporting the Light-weight Directory Access Protocol (LDAP) [7]. Such directory services are likely to become common place and even more robust in the future.

In addition to the above mentioned services, Metis requires printing and data-storage services, and a mechanism for launching and managing application-specific services on various servers. The latter can be accomplished, for example, using the servlets mechanism [2].

The rest of the paper is organized as follows. Section 2 presents the Metis thin-client programming model, sections 3 and 4 describe the Metis framework and implementation respectively. Section 5 discusses the related work, and Section 6 concludes the paper.

2 Thin-Client Programming Model

The key aspects of the Metis thin-client programming model are that the client downloads AFEs from the network; that these AFEs rely only on services found on network servers; that the services are bound as late as possible; and that AFEs interact with each other within the confines of a workspace. AFEs are securely installed and downloaded using a code service. They are also made 'thin' by implementing most of the application logic as one or more services. Late binding to these services provides:

- manageability, because services can be moved across server machines without impacting AFEs;
- flexibility, because services can be selected based on server load; and
- fault-tolerance, because a service can be obtained from an alternate server.

The workspace is a container for AFEs, allowing for interaction, as well as providing a shared environment. One important part of that environment is the authorization information that can be read from a smart card or provided as part of a logon process from an authentication service. The authorization information is used first to determine if a user is allowed to use the system, and then to identify the user's access rights to available AFEs. Afterwards, this information can be used directly by the AFEs to authenticate themselves to the service providers.

Figure 1 outlines the various building blocks of the thin-client programming model. It shows three important parts – the client workspace, application-specific services, and support services needed to provide full thin-client functionality.

2.1 Client Workspace

The client workspace provides a combination of functions in Metis. It provides the AFE container function, some of the conventional desktop functions, and a *virtual environment* of network services. These will be discussed in detail in this section.

Visually, the client workspace has a customizable layout that can be configured on a per-user basis using configuration information stored on a server. When a user logs on, all framework objects are instantiated.

UserProfile: The user profile includes an authorization object that contains user information including name and time of logon. The authorization object is passed with directory and code service re-

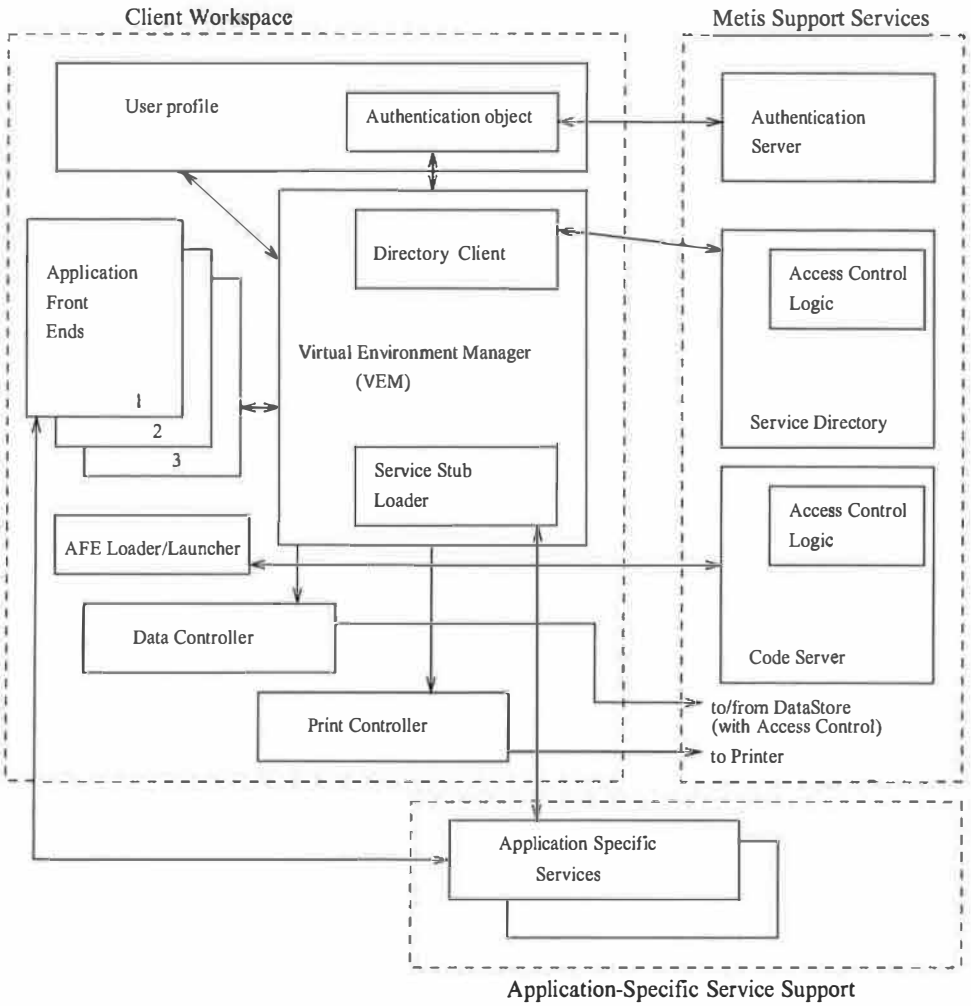


Figure 1: The schematic shows the three important parts of the thin-client application model: the client workspace, support services, and application-specific services. The client workspace contains user profile and authentication objects, objects to find and bind to services, and controllers for data and print. AFEs execute in the context of the client workspace.

quests. These support services recognize the object and restrict the user to only those AFEs and services that allow access by the user. Note that as long as services and AFEs are written to use the authorization object, a single logon procedure is possible.

Application Front Ends: The AFEs are downloaded to the client from a code server either when the workspace is initialized (if they were previously active) or when the user activates one on the workspace. They can be removed from the workspace as needed. Interactions among AFEs, such as data exchange and event notifications, are important especially when the AFEs are implementing sub-applications. Interactions are supported using *JavaBeans*^{TM} technology.

Virtual Environment Manager: A virtual environment manager (VEM) is a fundamental client object provided by Metis. It is the only entity with which an AFE can request Metis services. Prior to accepting an AFE's request, the VEM checks the AFE's signature to ensure that it is allowed access to the Metis system. As long as the AFE is recognized, the request is forwarded to one of the following VEM clients that act as delegates to Metis support services.

1. **Directory Client:** The VEM has an internal directory client that communicates with the Metis Directory Service upon request of a service from an AFE or the workspace. The directory client and Metis Directory Service together

provide late binding of services. At this time, services can be requested by name and service attributes though clearly, a higher level protocol can be supported. The directory client retrieves service stubs. If the service stub is an object, the directory client instantiates the object and passes a reference back to the caller AFE. If the service stub is a location, the directory client passes the location back to the caller AFE. To reduce the possibility of creating a flurry of messages, called a network storm, that can be caused by a failure of a widely-held service, the directory client retrieves and caches more than one service stub, when multiple providers of the same service are available. Should an active service fail, an alternate stub is fetched, instantiated if necessary, and returned to the AFE.

2. **Service Stub Loader:** The VEM has an internal service stub loader that communicates with the service providers to download any stub code that the service might need for providing the service. The service stub loader is only used if the service stub returned by the directory needs to be instantiated.

AFE Loader/Launcher: The Workspace has an internal object that communicates with the Metis Code Server to download code and launch AFEs. The AFE Loader also checks for digital signatures, uncompresses, and decrypts AFEs as needed. Enabling a client to download code from a centrally administered source makes its use attractive for intranet environments where the software distribution and maintenance on traditional PC clients is expensive. The AFE Launcher runs the AFE when it is selected by the user.

System Services: AFEs need a common set of system services such as printing, storage, and error logging. While application services are private to each AFE, Metis allows sharing of system services. When an AFE requests a system service, an associated controller object is returned. If a stub to the requested service does not exist, the controller creates one by accessing the Metis Directory Service to indicate where the stub class is. The class is downloaded, and instantiated. The controller, in turn, manages the stub instances. For example, in Metis a print controller is a single point of access used by all to 1) create access to specific printers and 2) send information to them. Most of the controllers are provided to support the AFEs. However, the data controller is also used by the Metis workspace to access user configuration.

2.2 Application-Specific Services

The Metis design imposes minimal requirements on service developers. They are free to implement services in any language. Communication between the service and the client-resident service stub can use any protocol, e.g., IIOP [5], RMI [4], or a private protocol. The service providers may provide a client-side stub with a well-known interface for access to its services, or, may only provide a location for AFEs to access services using a mutually understood protocol. The AFEs and service providers may use the authorization object provided by Metis for authentication purposes.

To better integrate services into the network, Metis provides a tool that can be used to register the services with the Metis Directory Service. For AFEs to dynamically access services, they must be registered with the directory. Registering a service makes it immediately available. Removing a service from the Metis Directory Service does not impact AFEs currently using the service. However, when an AFE detects that the service is no longer available, i.e., the service was removed from the server, it can fail-over to another service registered in the Metis Directory Service.

2.3 Metis Support Services

Metis has a number of server components performing distinct functions. While these services are not fundamentally part of the Metis thin-client programming model, they are needed to support that model. For example, the model states that AFEs can bind to any service available on the network that meets its requirements. To have the capability to find all such services, a directory service is used.

Metis Directory Service: The Metis Directory Service accepts queries from the directory client and sends results back to the client. It does not manage the physical service directory. Instead, the Metis Directory Service acts as a client to an LDAP [7] directory server. LDAP is an emerging standard in distributed directory services offering reliability and scalability. Each service in the directory has a unique service location and a number of attribute/value pairs. Each service must at least have a *name* attribute with a non-empty value. Services can be looked up by a name and a search filter composed of a boolean expression of attribute/value pairs. The Metis Directory Service can also perform access control via LDAP with the authorization ob-

ject that the directory client supplies. The Metis Directory Service allows service providers and code-server administrators to add, modify, or delete services in the directory.

The design of the Metis Directory Service simplifies ports to other directory technologies supporting both current and emerging network directory standards. It also can be easily enhanced to provide intelligence to the service selection process. As mentioned earlier, AFEs must currently know the given name of a service and important attributes as well as their correct values. For example, if an AFE wanted to access a color printer service, in the present design it would have to ask for one by name, e.g., ColPrt2. In the future, it might want to ask for a printing service that is physically close, e.g., nearby & color.

Metis Code Server: The Metis Code Service is an AFE repository. It provides central administration of client code and lends itself to be a *tuner* for a third-party code service that uses Marimba [8]. The Metis Code Server can perform access control using the authorization object. It can also digitally sign AFEs to identify them as part of the Metis system. Like the Metis Directory Service, the Metis Code Service is independent of the actual code distribution mechanism.

Metis Authorization Service: The Metis authorization process can either be completely self-contained, e.g., all authorization information is present on the user's smart card, or it can use a standard authorization service such as Kerberos [9]. A reference implementation for the Metis Authorization Service is being developed.

System Servers: Controllers provided by Metis act as the contact points for actual system services available on the network. In particular, a data-store service and printing service must be available on one or (preferably) more servers, and must be registered with the Metis Directory Service. These services receive requests from their associated controllers and return responses.

3 Metis Framework

The thin-client programming model described in the previous section recommends a general software architecture that allows applications to be written flexibly and to run within a manageable and fault-tolerant environment. However, to build these applications from scratch would be very difficult. The Metis framework, shown in Figure 2, presents Java classes and interfaces needed to make programming these applications easier.

The Metis framework provides instances of the key objects discussed in the programming model including the workspace, user profile, AFE loader/launcher, VEM, directory client, service-stub loader, and controller objects. It also manages AFE objects. In addition, interfaces are provided to assist the application developer with the task of writing a thin-client application. These interfaces can be grouped as follows:

- AFE integration into the workspace
- AFE access to services
- Metis System Services interfaces
- Metis Support Services interfaces

The following subsections discuss the interfaces shown in Figure 2.

3.1 AFE Integration into the Workspace

One purpose of the workspace is to contain and launch AFEs. A Metis user indicates which AFEs the workspace contains by using a "ToolBox" that provides the user with a list of all AFEs that he is entitled to use. When the user selects an AFE to add to the workspace, the Toolbox returns the location of the AFE class. The workspace passes the location to the code service client to download the class from the Metis Code Server. The icon associated with the class is accessed and added to the workspace as a button. To launch the AFE, the user double clicks the button. An AFE can be removed from the workspace container by removing the button. If it is active, then the AFE is destroyed. The workspace configuration is automatically saved when the user logs out.

Another purpose of the workspace is to provide resources to all AFEs. These resources include the user profile object and the controllers. To access these resources, the AFE is required to get a reference to the workspace object by implementing AFEInterface. When activating an AFE, the workspace calls the setWorkspace() method on the AFE, passing a reference to the workspace object.

3.2 AFE Access to Services

To access services, AFEs use the interface called the VEMInterface implemented by the workspace object. The workspace object delegates the VEMInterface calls to its member VEM object, that truly implements the VEMInterface.

To bind to a service, the AFE calls the request-

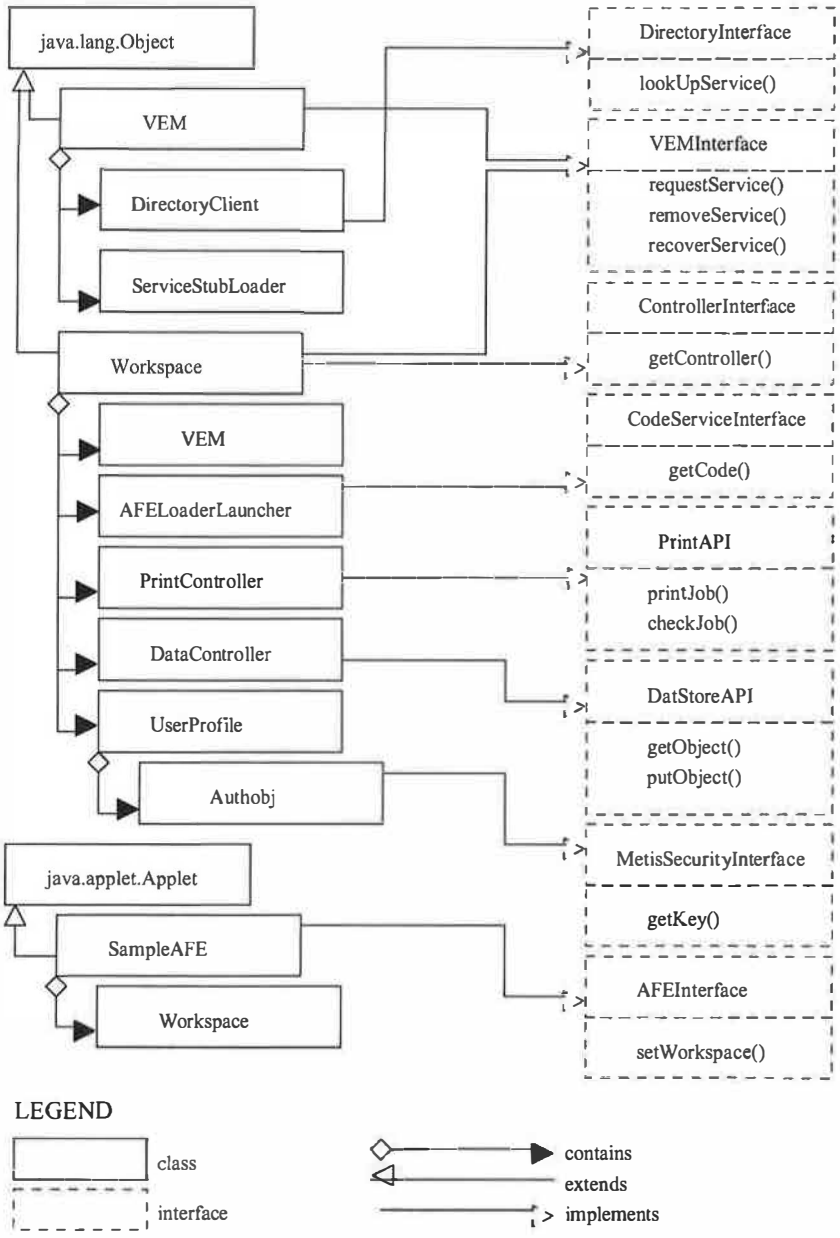


Figure 2: The figure shows important classes in Metis and their relationships. AFEs acquire a handle to the Workspace by implementing the AFEInterface. Workspace implements the VEMInterface and the ControllerInterface, and thus provides the necessary support for AFEs. Workspace uses delegation (to the VEM object) in implementing the VEMInterface. The workspace object contains VEM, controller, and UserProfile objects, and manages AFE instances.

Service() method on its workspace reference. The requestService() method takes the AFE instance, the name of the service, and a preferred list of attribute/value pairs as arguments. A Metis object, called a ServiceInfo, is returned that contains an instance of the service stub or, if desired, the service location. All information needed to access the service is now available to the AFE. Should a service fail while being used, the AFE can re-request a new service by calling the recoverService() method on the workspace, passing a reference to itself and the active ServiceInfo object.

3.3 Metis System Services Interfaces

Metis defines a generic Controller class for managing instances of objects that provide shared system services to all AFEs on the workspace. The Controller class provides a graphical user interface for adding and removing system-service instances from the workspace and for allowing the user to select a specific instance to provide the service. Service-specific controllers extend the Controller class and implement service interfaces to provide the desired service. The services are explicitly configured using a GUI provided by the Controller. A user can also setup a specific instance of the service provider as a default for that service.

To access the system services, an AFE calls the getController() method. For example, should the AFE need access to the data store, it calls the getController() method, passing it the type of system service required, e.g., 'DATASTORE'. The VEM returns the instance of the appropriate controller for that service. The AFE then uses the instance when it needs system services. The Controller in turn, delegates the request to the default service provider, or allows the user to select a service provider from the list of configured providers.

Metis also provides reference APIs for system services such as printing and data storage. The printer API allow an AFE to print, check the status or remove a print job. The data storage API allow an AFE to store and retrieve files from a data store.

3.4 Metis Support Services Interfaces

The Metis Support Services interfaces are not visible to the application programmer. They are hidden within the VEM, controllers, and the user profile. At this time, three key interfaces have been identified. The DirectoryInterface has been designed and implemented. The CodeServiceInterface and the MetisSecurityInterface are being developed.

The directory client uses the DirectoryInterface to avoid being tied to a single directory protocol. Any physical directory service can transparently plug into Metis simply by enhancing the Metis Directory Server with updated implementations of the DirectoryInterface. The interface has been kept small, containing only one method, to minimize the complexity of directory access. When the VEM client is requested to access the Metis Directory Server (e.g., an AFE called the requestService() method), the VEM calls the lookUpService() method on the directory client. The authorization object, name, and attributes passed to the VEM are passed as arguments to the lookUpService() call.

4 Metis Implementation

This section gives the highlights of the implementation for both the Metis framework and the AFE suite used to demonstrate it. The discussion of the Metis framework implementation will be restricted to the workspace implementation and the service access interfaces, i.e., the VEMInterface and the DirectoryInterface implementations, along with important utility objects that are used. The discussion of the AFE implementations will center on what specifically was done to integrate the applications into the thin-client programming model. Overall, the Metis framework is about 7000 lines of Java code.

4.1 Metis Workspace

The reference Metis implementation supports Java bean behavior, to allow a natural mechanism for AFEs to interact within the workspace. The Sun Microsystems reference implementation of the BeanBox is the basis of the Metis workspace. The BeanBox provides bean containment, and support for property sheets, visual manipulation of beans, and state storage.[1]

At workspace startup, a logon process verifies the user and creates the user profile object if the verification is successful. This process includes showing a dialog box for user name and password information and checking it against an authorized user list. The user is also allowed to use a "smart card"[2] instead of providing information directly to the logon dialog. If the user passes verification, the VEM and controller

[1] Unfortunately, some of these functions (e.g., event interactions, property sheets, and pickling) had to be disabled to get the base functions working since the version of JDK1.1 (Beta 3) was unstable. These functions will be reintegrated as JDK1.1 gets more mature.

[2] A floppy drive was used as a place holder to smart card hardware.

objects are instantiated.

In Metis, the workspace is populated with buttons, each showing the icon of the AFE. Each button contains the URL of an applet and an appletviewer instance with a reference to the applet. The appletviewer has been modified from the JDK implementation to separate the applet instantiation from its execution. When a button is selected, the appletviewer initiates the applet execution. The initial layout on the workspace is read from the user's configuration file, accessed from a data store.

The ToolBox has been completely rewritten from the BeanBox implementation. In our implementation, the ToolBox queries the directory server to get a list of AFEs that the user can access. It allows a user to add an AFE as a button to the workspace, and when the button is added the AFE loader downloads the AFE code from the Metis Code Server.

4.2 VEMInterface Implementation

An AFE requests a service by calling the requestService() method on its workspace reference. The AFE passes a reference to itself, the well-known name of the service, and a special Filter object. Passing the requester of the service enables the VEM to keep state on each AFE. It also allows application designers to, in effect, build a service providing object that gets services on behalf of all AFEs in the suite. The Filter object was designed to simplify the attribute/value logic specification. At this time, only the "AND" function is provided, e.g., a service must have "attr1=val1" & "attr2=val2". The Filter class will be enhanced in the future to allow arbitrary logic specifications. An example code segment showing service request is:

```
Filter filter = new Filter();
filter.addElement("height","low");
filter.addElement("speed","slow");
workspace.requestService(this,
    "my_service", filter);
```

The requestService() implementation in the VEM class does the following list of actions. Note that not all exception paths are included in the list.

1. If a null filter was passed in, the VEM creates one. A special attribute/value pair ('type'/'Service') is added. The type attribute was added because the Metis Directory Service contains both services and AFEs.
2. The VEM checks its internal state to see if the AFE is a known service user.
3. If the AFE is not a known service user, its authorization is checked to ensure that the AFE has the credentials needed to use the Metis system. If it passes the check, it is added to the VEM internal state. Otherwise, the requestService() throws an exception.
4. The VEM checks its internal state to see if the AFE has previously asked for the same service. If so, it returns the associated ServiceInfo object.
5. The lookUpService() method is called on the directory client and the returned list is wrapped in a ServiceInfo object.
6. The VEM checks the validity of the ServiceInfo object. The object will be invalid if the directory search could not find any services, e.g., if the service was unknown or if the user did not have access to the those registered. An exception is thrown if the ServiceInfo is invalid. A timestamp is also added to the new ServiceInfo object. The ServiceInfo object is then added to the VEM internal state and returned to the AFE.

The object class, ServiceInfo, returned by the requestService() method maintains the information returned by the directory lookup. For the reference implementation, the locations of the services are stored within the ServiceInfo object as URLs. The class provides accessor methods for AFEs to use.

The AFE uses the ServiceInfo object to get to the service. During its use, the service could fail. The AFE, when it detects such an occurrence, can request a replacement service by calling the recoverService() method on its workspace, passing it a reference to itself, and the ServiceInfo it used to get the previous service. The workspace delegates the request to its VEM object. The VEM class's recoverService() implementation does the following list of actions. Again, not all exception paths are included.

1. Add the currently accessed service to the blacklisted services maintained in the ServiceInfo object. This blacklist is simply a means of tracking which of the services returned by the directory lookup have been used, and failed.
2. See if there is another service known in the ServiceInfo that has not yet been used. If so, setup

that service, make it the current service in the ServiceInfo object, and return back the same ServiceInfo object.

3. Otherwise, save the timestamp and the current blacklist. The timestamp is important in later steps. The blacklist is needed because more services may be available than those returned during the previous directory lookup(s). That is, the blacklist transcends the partitioning done by the lookup; It is used to capture the AFEs access to ALL relevant services.

4. Request the directory client to do a new lookup, passing it the blacklist, and wrapping the return list in a new ServiceInfo object.

5. If the ServiceInfo object is valid, then there were other services as yet unused by the AFE that met its requirements. Copy back the timestamp and the blacklist. Return the ServiceInfo object to the AFE.

6. If the ServiceInfo is invalid one of two scenarios could have happened. First, the current sweep through all possible services may have finished. If so, a new sweep is started by clearing the blacklist and creating a new timestamp. Second, there may be no more services available, even though they may still be registered with the directory service. One way for this scenario to happen is if there is a network partition that does not affect access to the Metis Directory Service but that interferes with access to the servers that the services are running on. It is detected by noting that the time required to finish a sweep is less than a pre-configured time determined by the system administrator.

7. If no new services are available, VEM repeats steps 4 through 6 one more time. If, even after the retry, it cannot satisfy the request then it displays an error message.

4.3 ControllerInterface Implementation

Controllers that implement the ControllerInterface provide AFEs with access to system services. In the current version, two types of controller classes are available: PrintController and DataController. The PrintController provides access to printers while the DataController provides access to data stores. Note that there is only one instance of the Controller for each system service. The interface consists of the method below.

```
public Object getController(int type) {
  switch (type) {
    case ControllerInterface.PRINTER:
      return printController;
    case ControllerInterface.DATASTORE:
      return DataController;
    default:
      return null;
  }
}
```

4.4 DirectoryInterface Implementation

The DirectoryInterface need only define the following method:

```
public SearchResult
  lookUpService(Authobj authobj,
       String name, String filter,
       URL[] blacklist, int howmany,
       String attrlist)
  throws java.rmi.RemoteException;
```

The requestService() method of the VEMInterface calls this method to access the LDAP directory. The parameters passed to this method are:

authobj: The authentication object from the workspace used for access control in the LDAP directory.

name: The name of the service being looked for.

filter: A filter composed of attribute/value pairs.

blacklist: A list of service locations that the client specifically does not want returned even if matched.

howmany: A count of matches to be returned.

attrlist: A list of attributes that are to be returned with every match.

The method returns an object containing service locations and attributes. The implementation of the lookUpService() method on the Metis Directory Server is straightforward. The Metis Directory Server acts an LDAP client. It constructs an LDAP query, and submits the query to an LDAP server. If desired, a random subset is chosen from the services returned by the LDAP server and are returned to the client. At the present time, the Metis Directory Server uses an LDAP client API. The use of the Java

Naming and Directory Interface (JNDI) [2] is being investigated.

The interface is simple but flexible enough to be used for various purposes. In addition to looking for service names and services with specific attributes, it can also return a list of all services (subject to access control) when used as:

```
SearchResult s=lookUpService(authobj,
    null, null, null,
    ALL_POSSIBLE, null);
```

where the null values indicate no filtering and subsetting is to be performed; or, can return the count of all services with the name "LotusNotes" when used as:

```
int i=lookUpService(authobj,
    "LotusNotes", null, null,
    ALL_POSSIBLE, null).getCount();
```

4.5 AFE Implementations

A number of applications were implemented to demonstrate the usefulness of the framework, including an application (called ViewGlass) that can access Lotus Notes servers and a financial application suite. In this section, the AFE/service partitions are described as well as AFE use of the VEMInterface to access its service for the ViewGlass application.

4.5.1 ViewGlass Implementation Notes

ViewGlass provides a user with the ability to access her Notes mailbox (to send, read, and delete messages), and to read discussion databases. The functional split for this application is that the AFE, written in Java, would contain the GUI, rich text browsing support, and a private communication layer that directly accesses a proxy service. The service, written in 'C', would accept messages from the GUI, call the appropriate NotesAPI functions, and return information back to the GUI. The service runs as a daemon and contains client-specific state.

During the ViewGlass initialization process, the requestService() method is called to get the location of a known Lotus Notes service. The location of the service is then accessed and a socket connection is made. This is shown in the code below.

```
try {
  sinfo = workspace.requestService(
  (Object)this,"LotusNotes",null);
} catch (Exception e) {
  System.out.println(
  "LotusNotes service not found");
  destroy();
  return;
}
URL url = sinfo.getURL();
server_name = url.getHost();
server_port = url.getPort();
```

Connection failures are notified to the AFE via the exception mechanism. The AFE calls the reset() method to recover from failure.

```
public void reset() {
  try {
    sinfo = workspace.recoverService(
        (Object)this,sinfo);
  } catch (Exception e) {
    System.out.println(
    "LotusNotes service not recovered");
    proxy_recv = null;
    destroy();
    return;
  }
  URL url = sinfo.getURL();
  String server_name = url.getHost();
  int server_port = url.getPort();
  // ...  make the connection
  wk_sp_frame.recover();
}
```

Note that the reset() method ends with a call to the recover() method. Since some client state is maintained in the service, full recovery is not possible from just the client. However, the state completely contained in the client is recovered.

5 Related Work

During the last year, many players in the computer industry have focused attention on alternatives to the traditional client. One of the design points for most of the efforts is to ensure that the users continue to have access to all resources that they are accustomed to. One well-investigated system is to provide access to applications using a browser. In this system, the application runs mostly on well-known servers. HTML pages, some enhanced with small Java applets, are sent back to the client. The browser relies on an underlying operating system to get access to files and printers. The browser metaphor works well if there are only a few applications that do not interact much and if each application has a restricted amount of user interaction.

The browser system is one possible choice for enabling the use of Network Computers as the hardware client for thin-client applications. However, for environments where an application is composed of many sub-applications that may interchange data frequently, the browser metaphor is not sufficient. The browser metaphor also lacks the integration of network-based access to system resources. Other alternatives like defining Network Computer desktops or webtops (i.e., the Lotus DeskTop and HotJava Views) have been investigated as extensions to the browser metaphor. Also, many vendors are vying to provide environments for Network Computers to access system resources commonly found on traditional desktops, like printing and file access [10].

During the Metis effort, we emphasized enabling commercial applications for Network Computers. Many of these business applications are currently single-system based. Moving to a network (1) introduces unreliability not often found with standalone systems, (2) raises security concerns, and (3) distributes resources like printers and files. While the browsers and Network Computer desktops could handle the latter two issues in future implementations, they are not meant to address the first. Both the Lotus DeskTop and HotJava Views could be integrated with Metis by replacing the workspace used in the reference implementation, to address all three issues today. Metis would then also provide a framework for developers to implement robust applications for browsers and Network Computer desktops.

The previous paragraphs focused on browsers and desktops that provide access to complex applications for Network Computers. However, Metis provides a distributed application technology as well as a user system. There are several distributed application technologies for traditional clients and servers that some developers could use for Network Computers. These technologies include CORBA and design patterns.

CORBA [11, 12, 13], is a distributed application technology specified by OMG that emphasizes reusable services and facilities. These specifications allow applications to interact with other application modules independent of the machine architecture and language the modules have been written in. OMG's Trading service specification allows an application to query and identify service names that match a particular criteria. These service names are then bound to a particular object as per the Naming service specification. The combination of the two services can be used by applications under CORBA to achieve late binding of a name to an object. Metis provides similar late binding of service providers to an AFE using an LDAP directory server. AFEs can specify service properties through the Filter class. Results of the match are available to the AFEs for binding and use.

Design patterns [14, 15] have been proposed as a technique for application development that also emphasize reuse of software architectures, including those for distributed systems. Design patterns allow software developers to write their applications using high-level models that are independent of language and machine architecture. The patterns focus on key components and their interaction to facilitate reuse of software. Design patterns have been used for writing large scale commercial applications. The Metis workspace has been written as a design pattern for desktops on thin clients. The workspace provides components for locating and binding to services, access to system services, and security components. The intent is to allow application developers to use the workspace pattern in developing application suites for thin clients.

6 Conclusions

In this paper we presented a thin-client programming model where clients download application front ends that have a presentation layer and some application logic, but the bulk of an application is executed as services on remote servers. We described the design and implementation of a framework, called Metis, that enables the thin-client programming model, and showed how it can be used in sample applications.

The design of the Metis framework has an open architecture composed of abstract interfaces for various services so that any implementation can be plugged in. We implemented both client- and server-side infrastructure and realized a full end-to-end framework that provides support for:

- finding and binding services
- access control and authentication
- system services
- code services

Metis support for late binding makes it possible to write reliable, flexible, and manageable thin-client applications. Moreover, the Metis framework provides true platform independence beyond the language level by virtualizing all system resources as services. A demonstration of the reference Metis implementation that includes the Metis classes and

API documentation is available at the IBM alphaWorks web site (http://www.alphaworks.ibm.com) as TCAF.

During the course of this work we identified several areas of further research which may be beneficial to the thin-client programming model. These include support for application-specific recovery, remote event mechanisms,and improved security and communications. We plan to explore the above areas as we enhance Metis to fully realize the benefits of the thin-client programming model.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[2] Javasoft. *Javasoft Home Page.* URL = http://www.javasoft.com/, 1997.

[3] Javasoft. *JavaBeans Component APIs for Java.* URL = http://splash.javasoft.com/beans, 1997.

[4] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the USENIX Conference on Object-oriented technologies*, pages 219–231, 1996.

[5] OMG. *CORBA/IIOP.* URL = http://www.omg.org/corba/corbaiiop.htm, 1997.

[6] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(5):22–35, June/July 1988.

[7] Timothy Howes and Mark Smith. A Scalable Deployable Directory Service for the Internet. In *Proceedings of INET 95*, 1995.

[8] Marimba, Inc. *Marimba.* URL = http:// www.marimba.com/, 1997.

[9] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[10] Novera EPIC Release 1.0. Novera Epic Developer's Guide. Technical Report EPICDEV-100-1, Novera Software, Inc, November 1996.

[11] OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, 1995.

[12] OMG. *CORBAservices: Common Object Services Specification.* OMG, 1995.

[13] OMG. *CORBAfacilities: Common Facilities Architecture.* OMG, 1995.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[15] Douglas C. Schmidt. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM*, 38(10):65–74, October 1995.

Frigate: An Object-Oriented File System for Ordinary Users*

Ted H. Kim Gerald J. Popek†
Department of Computer Science
University of California, Los Angeles

Abstract

Vendors cannot provide all the operating system services that users demand. As a result, there has been a persistent desire to make operating systems more flexible and customizable. It is natural that object-oriented technology would come to bear on this area. However, many solutions have been disappointing when it comes to ease of use.

This paper describes the design and implementation of Frigate, an object-oriented file system. The goal of Frigate is to provide a modular, extensible framework. The framework allows new extensions to be "plugged-in" on the fly. Frigate's focus differs from most other file system designs in that it is targeted for use by *ordinary users* rather than by sophisticated operating system gurus. Thus, ease of use is a very important concern in the design. Frigate is fully implemented and supports a set of example file system extensions.

1 Introduction

Vendors cannot provide all the operating system services that users demand. On the one hand, it is economically infeasible, and on the other hand, many times they do not know what the users desire. Vendors rightly concentrate on providing a few general purpose services. Users can either lobby the vendor to include some desired service or implement it themselves. Adding services to the operating system is a difficult proposition. Traditionally, special privilege and access to source (and possibly a license agreement) is required. Programming the kernel can demand special care and knowledge. Debugging modified operating systems is difficult as well usually requiring specialized kernel address space tools as well as continual rebooting. Convincing someone else that your extension of the operating system is not too risky is not always easy. Distribution of modifications may encounter similar trust problems and be limited by license restrictions. Compatibility between independently written extensions and the ability to further modify them are also potential problems.

The long standing desire to make operating systems more flexible and customizable led to various architectural innovations, including loadable device drivers, streams [42], vnodes [28] and micro-kernels. All of these ideas were aimed, at least in part, at providing extensibility. It is natural that object-oriented technology, with its themes of modular extensibility, would eventually come to be applied to this problem.

Most attempts at applying object-oriented technology to operating systems are attempts to internally restructure the operating system into a more modular organization. The object-oriented model is not generally exported to the users. The intended audience of such solutions are expert operating system architects, who are conversant in the intricate internals of the operating system. Such tools are extremely hard for ordinary users to use.

Our particular problem domain is the file system. Within this domain, Frigate takes a different approach. The intended audience of Frigate is ordinary users. Frigate's object model is not just for internal use. It is fully exposed and usable by ordinary users. A programmer using Frigate needs to be familiar with object-oriented concepts and system-call programming but does not need to be an operating system guru. We believe we have constructed tools that can be easily and widely used. This enables a much larger audience to do powerful, modular extensions of the file system.

Our overall goal is to provide a way to add value to the file system easily. Towards this end, Frigate attempts to provide a modular, easy-to-use, persistent object framework that also allows incremental usage and is fully compatible and integrated with the current filing environment.

*This work was supported by the Advanced Research Projects Agency under contract DABT63-94-C-0080. The authors can be reached at the Department of Computer Science, UCLA, Los Angeles, CA 90095, or by email to {tek,popek}@cs.ucla.edu.

†Gerald Popek is also affiliated with Platinum *technology*.

2 Architecture

The Frigate system consists of five main components built on top of UNIX. At the lowest level, Frigate uses *typed* files. File System extensions are stored in an external *Repository*. When extensions are used, they are instantiated as server processes. Within the operating system, Frigate provides a *Dispatcher* module, which manages the servers and intercepts file system calls, passing them out to the servers. This module is built on the Stackable Layers framework. Finally, an object-oriented programming interface is provided by using Xerox PARC's Inter-Language Unification (ILU) system [25]. The runtime architecture of Frigate is illustrated in Figure 1.

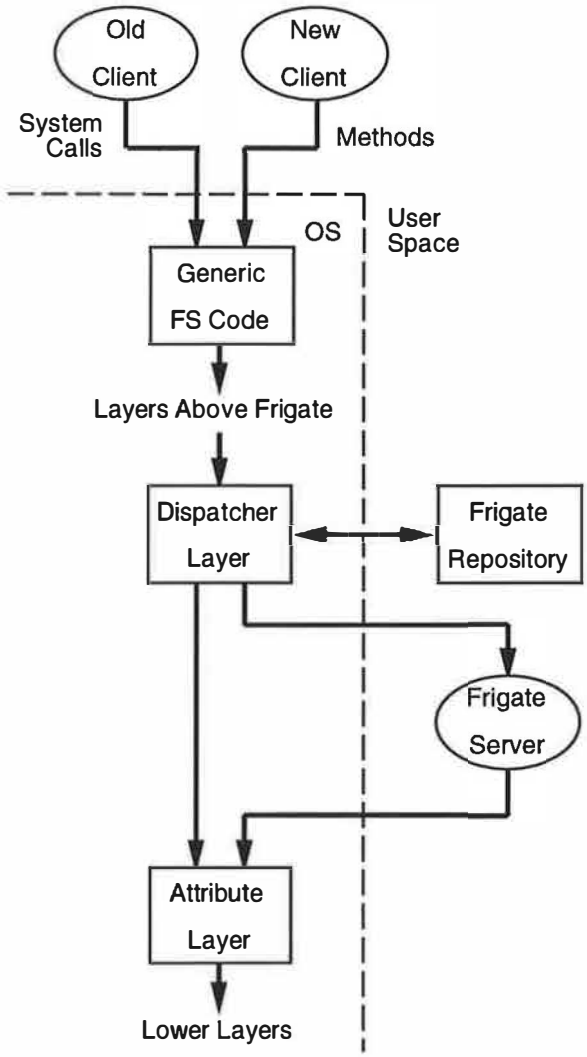


Figure 1: Frigate Architecture

2.1 Stackable Layers Infrastructure

Frigate marries two different worlds together; it connects a UNIX file system model with a CORBA [36] based object model. On the UNIX side, Frigate uses the Stackable Layers framework, which can be seen as a generalization of the Virtual File System (VFS) [28]. VFS is the basis for most modern UNIX file system implementations. In VFS, the file system portion of the operating system is divided into a generic portion and specific file system implementations (e.g., Unix File System (UFS), NFS, PC-FS). All files (directories, pipes, etc.) are represented in VFS by an abstract type called a vnode (virtual node). All calls into the specific implementations are operations on this abstract type. This allows other specific implementations to be added without any modification of the generic code. In practice, though, this is difficult as each specific implementation is still quite large and must be developed and debugged in the operating system address space.

A further refinement of VFS is the UCLA Stackable Layers file system [23]. In this model, each specific implementation is made up of a stack of protocol layers in the kernel similar to System V streams [42]. Each layer is a much smaller entity than the large monolithic file system implementations of pure VFS. Generally, layers implement a specific increment of file system features. Operations not implemented by a particular layer are passed down to lower layers in the stack, via the "bypass" operation (a form of delegation). The layers use the vnode interface for inter-layer calls and can be independently developed, replaced and composed together. For example, a file system might be composed of a layer for naming/directory services and a layer for storage. Later, the storage layer could be replaced with one implementing a log-structured storage [43] without requiring any changes to the naming layer. Stackable Layers was used to implement the Ficus replicated file system [18, 19]. Layers need not be configured strictly linearly as stacks but can also be placed in tree configurations. As we will see, Frigate extensions are implemented in this model as layers.

2.2 Documents

Frigate uses typed file entities, which we call *documents*. A file entity is anything that can be named in the file system: file, directory, pipe, etc. Along with the standard UNIX file attributes, each document has associated class and version identifiers. The class is a name for a particular method interface

and hence a set of services provided by the document. The version is used with the class to pick a particular implementation at *bind* time. To prevent name clashes, but at the same time, permit independent development, the actual identifiers are NCA UUIDs [52], which are globally unique but can be generated in a distributed fashion. This obviates the need for any central registry. In our current implementation, the storage for these attributes is provided for in a separate Stackable Layer [50].

Documents are objects in our system. However, they have some distinctive features. In our model there is a standard implicit UNIX file class, which all document classes inherit from directly or indirectly. Vnode operations appear in the Frigate model as methods. Document classes can declare that their associated versions have modified vnode method implementations. This redefinition of vnode methods is possible for all file operations, except for a few reserved for Frigate's infrastructure. Of course, document classes can add any other methods also required for their application usage. Because documents have an associated UNIX file entity, they can be named by using file pathnames and can also use file data storage.

2.3 Repository

The repository stores the interfaces and implementations for Frigate extensions. Each class has an interface description indexed by class identifier. The interface description consists of identifiers for vnode methods that are redefined by the document class. This information is used by the Dispatcher to determine if a vnode operation should be intercepted and passed to an extension server. The Dispatcher also sends vnode operations carrying ILU method invocations to the servers.

Implementations stored in the repository are the actual servers that implement objects. The implementations are indexed by class and version. One implementation for each class may be marked as *preferred*. Applications can specifically request the preferred implementation. A preferred implementation is usually the latest or most stable version. Stored with the implementation is load balancing information and configuration parameters for the server.

The repository also supports *alias* records. An alias is an alternate name for an implementation or another alias record. It defines a mapping from one implementation identifier to another identifier. As described later, this feature is used to aid in version management.

The repository is implemented by a repository server and database. Requests from both the Dispatcher and front-end user programs are handled by the server. The repository server handles various requests from the Dispatcher involving server management, interface query and binding of implementations. The process of binding involves mapping a class/version pair to a specific server implementation, taking into account preferred implementations and alias records.

Front-end user programs interact with the repository server mainly to add and remove class descriptions, server implementations and alias records. Security features only allow the class owner to manipulate the class description and associated implementations. Otherwise, all users are capable of adding and managing classes and implementations.

2.4 Dispatcher

The Dispatcher is a stackable layer that stacks in the operating system above the layers that provide file storage and Frigate attributes. This layer intercepts vnode operations for documents and directs the management of the servers that implement document objects. From the view of Stackable Layers, this is just another layer. The Dispatcher layer, however, does not have the same behavior for all files. Instead, the behavior of the Dispatcher varies according to the class and version of the document.

When the Dispatcher vnode for a newly referenced file is constructed, some special actions occur. First, it is determined whether the file is a document or not. A document has Frigate attributes; other file entities do not. If the file entity is not a document, the vnode is set up to simply bypass all operations to the layer beneath it. Thus, ordinary files behave as if the Dispatcher layer did not exist at all.

If a vnode belongs to a document, then the class attribute is used to find its interface by querying the repository server. (Information is cached, so subsequent queries may avoid RPC with the repository server.) The interface information from the repository is used to determine which vnode operations should be intercepted and sent to the server. All other operations bypass to the layer beneath.

Servers are lazily started; no attempt is made to start one until some method or vnode method is called. In this way, no unnecessary server starts are performed. When an operation that requires a server is invoked, the document is actually bound to a particular implementation. This late binding allows the most recent repository changes to be reflected in the binding, even if these changes occurred since the time of class interface lookup. When the

actual implementation is determined, the list of active servers is checked for a matching server. If a matching implementation server has spare capacity, then the document is assigned to that server. If no server is available, then one is started.

After a document has been assigned to a server, methods and vnode methods are passed to the server using Stackable Layer transports. Methods pass through special vnode operations that carry ILU information to and from the server. A server may handle requests from several different users. For the duration of each request, the server's credentials are changed to that of the process that invoked the method. Thus, the server will have exactly the privileges of the client. Between requests, the server is given special reduced privileges. If we did not have such a mechanism, it would not be possible to have a single, possibly *untrusted*, server handle multiple users. Each user would have to have his own server to assure correct access credentials.

When a document is no longer referenced, the vnode for the document is destroyed. During this process, a message is sent to inform the server. When a server no longer has any documents assigned to it, the Dispatcher directs the repository server to terminate the server.

2.5 Servers

Stackable Layers provides some facilities to run user-level file system servers. Frigate servers are modified and enhanced versions of the Stackable Layer servers. Frigate servers are structured somewhat differently to allow modular construction from ILU files and to accommodate method dispatch at runtime. Since Frigate servers are user-level, server programmers do not have to worry about kernel programming restrictions. Frigate provides the means for running more than one implementation side by side for any particular class. Also, multiple servers can run for load balancing.

Documents are persistent objects, which can be in one of two states: *active* and *passive*. An active document has been assigned to a server and is ready to service requests (i.e., has a running implementation). A passive document has no server. Whenever a document is newly assigned to a server, methods are automatically invoked on the document to activate it. Typically, these methods initialize the document's implementation from its persistent file data. When the last reference to a document is removed, methods are automatically invoked to deactivate the object. Usually, these methods force any pending updates out to persistent storage.

The actual implementation of a method lies in its handler code. If necessary, the handler may call on services from layers lower down in the stack. Calling on lower layers leads back into the operating system rejoining with the in-kernel stack. This is accomplished by using a special Stackable Layers transport.

2.6 Programming Interface

The Inter-Language Unification (ILU) system [25] from Xerox PARC is used by Frigate to provide an object-oriented programming framework. ILU provides a CORBA [36] based object model. In ILU, type definitions and interface definitions for objects (including multiple inheritance relationships) are written in CORBA IDL (or ILU's own ISL). These definitions are compiled into target language mappings, providing client calling stubs and server method skeletons. The target language mappings are compiled along with application code to construct the actual ILU client and server programs. At runtime, ILU clients and servers communicate by using transports selected from ILU's library.

Frigate integrates document objects and vnode methods with ILU. Currently, Frigate implements support for documents in both IDL and ISL and provides support for the ANSI C target language. Frigate applications are programmed just like other ILU applications, except that servers must be assigned a version number and registered with the repository to be used. Compiling interface definitions automatically produce the required calling stubs and skeletons for any vnode methods. Server method handlers may be programmed using the Stackable Layers vnode model or the regular UNIX system call model.

Frigate document classes appear in the ILU world as ILU object types. They are distinguished from other ILU objects by inheriting object types from the implicitly defined UNIX file interface. This avoids any syntactic change to ISL or IDL. Only documents may have vnode methods. The semantics of inherited vnode methods, though, are different from those of other inherited methods. Even if a particular vnode method is not inherited, a document will still try to delegate that method invocation by using the Stackable Layers bypass mechanism. Vnode methods are, by design, inherited only. The signatures of vnode methods are externally fixed and so cannot be redefined inconsistently with the rest of the framework.

At runtime, Frigate documents are located by using file system paths. The file system namespace acts as the published object registry. Through the

use of Frigate utility routines, the file system name of a document is resolved internally into an object handle and then into a local proxy object. Thereafter, messages for ILU methods or vnode methods can be sent to the document object by using the calling stubs defined in the language mapping. Method invocations made on proxy objects are automatically converted into calls to the Frigate document servers. The calls are transported to Frigate servers via the Frigate transport, which packages the Stackable Layers infrastructure as an ILU transport.

It should be noted that vnode methods can be invoked both through ILU and through ordinary UNIX system calls. UNIX system calls are serviced by the VFS generic file system code, which in turn makes vnode operation calls. These calls are selectively intercepted by the Dispatcher just as direct vnode method calls would be. This means that Frigate document servers can even enhance the behavior of programs that have no knowledge of the Frigate object model by enhancing standard UNIX file operations. Thus, almost all existing UNIX programs can transparently use Frigate.

2.7 Versions

The mechanisms provided by Frigate allow for multiple models for version management. In all cases, the repository accommodates installation of classes and implementations without any disruption of system operation. Currently active documents will continue to execute with their bound implementations; any new use will immediately reflect any repository updates. Multiple implementations for the same class can run side by side. New implementations can be installed and tested in isolation before general use by proper version management.

Documents marked with preferred versions always use the implementation so marked. This feature is used to support an "eager" model of version management. In this model, the newest accepted version is designated as preferred. In this way, documents can automatically use the most recent version. New versions, though, should not be designated as preferred until tested sufficiently.

A more selective model uses aliases to effect the update. After acceptance of a new version, the old version is replaced by an alias that maps to the new version. Any documents with the old version identifier automatically use the new version. There is no need to hunt down existing documents and update their version identifiers. As documents are used, they *may* have their version identifiers rolled forward. However, it is not necessary for *all* old versions to be replaced by aliases. Situations where there are incompatible changes in data formats or limited trust placed in a new version can be handled. This is a "lazy" model where no change is made until an old version is explicitly mapped to the new version. It is also possible to have chains of aliases that converge to allow future coalescing of versions.

In many object-oriented systems, subclasses are often defined to provide a specialized implementation even when there is no interface specialization. For example, an image object class may have subclasses for GIF and TIFF formats. Since implementation in Frigate is completely divorced from class definition, the version system can allow specialized implementations *without* subclassing. No duplication of interface is needed. Separate specialized implementations can be installed under the same class. Thus, two different versions do not necessarily represent different stages along a single line of development. They may, in fact, be parallel separate lines of implementation. In Frigate, there may be GIF and TIFF implementations for the same image class.

The version system can, in effect, provide a form of polymorphism. The advantage of doing this is that simple repository commands can give the user complete control over the binding process. By using aliases, the evolution along any particular line of development can still be transparently managed. If future implementations merge lines of development, aliases chains can be made to converge as well.

3 Examples

Frigate has a variety of possible uses. Here we outline some example possible applications of Frigate to illustrate the flexibility of the framework. We have actually implemented a transparent encrypted file system, image files, intentional files and a simple access control system.

Frigate can be used to provide file system features such as transparent compression, encryption and migration. For the most part, these are services that continue to use the ordinary system call interface. In Frigate, such services are provided transparently and automatically by redefining vnode methods. The new definitions for vnode operation enhance the behavior of standard system calls. In this way, no change to old programs is necessary to take advantage of enhanced behavior.

Another example of this type is the *intentional* file. Instead of explicitly storing the contents of a file, the contents are instantiated on the fly. Traditionally, UNIX stores information on users, hosts and networks in a series of files. Sun's NIS service modifies

the standard library calls to access network servers instead of the files. Frigate can provide the same sort of service through intentional files. The file contents are generated on demand based on information from the network servers. The advantage here is that even programs that have the old library code can benefit. Dynamic information, such as network loads, logs or realtime data can similarly be provided by an intentional file interface. Intentional files can also be used as a form of compression for large but easily regenerated datasets.

Other applications might use a mode of operation that is not strictly transparent. In such cases, the interface is extended to provide additional functions. An example is an enhanced file access control system. In such a system, there are additional methods to manage access control lists, provide different levels of information to users, read logs, etc. However, the majority of the file system access (e.g., read/write) still goes through the ordinary system call interface.

Frigate also provides an object-oriented model of access. Programs using this model can take advantage of polymorphism. A single document class for image files might have several implementations, each of which services a different image format. A single program could be written to manipulate any of the image files via the document interface. The program would not have to be changed, if available formats changed or if a format was added after the program was written. This is because the implementations (servers) are completely separated from the interface (class definitions) and from the client program binary. Despite the generic nature of the program, format specific code is executed for each particular file for efficient access.

The UNIX file interface is limited to addressing file entities. Frigate's object-oriented model can address other sorts of objects. Messages may be sent directly to parts of a compound document represented by objects. This allows compound documents similar to those in OLE [35] or OpenDoc [14] to be constructed in Frigate. In this case, a document object is a container for other "content" objects, which contain text, images, audio, spreadsheet cells, graphs, etc.

Frigate's facilities can be used to provide *links*. Links are potentially interfile, persistent references to content objects that can be stored in documents. (ILU object handles are not persistent.) With a link tracking service, updates of a source can automatically be reflected in any document containing link references. Frigate can also be used to provide more general models of interfile consistency. Changes in one document can be automatically propagated, appended, merged, etc. into other documents.

UNIX file operations are low level. Frigate can provide much higher level file system abstractions. Documents might have operations to print, archive, install and move (with all other associated objects). Operations can be tailored to be more meaningful to the application. Documents are capable of any action that can be accomplished by a program, because documents are implemented by server programs. For example, a document might automatically enforce integrity constraints or warn other users of important changes by email.

4 Extensibility

Our overall goal of making the file system easier to extend has several aspects. We provide for extensions that are not resident in the operating system address space. This frees us from the classic problems of operating system development including: privileged access, source access, specialized operating system knowledge, programming and debugging restrictions, rebooting, potentially disastrous effects of bugs and redistribution restrictions.

Frigate is designed to scale. Frigate class and version identifiers are assigned in a distributed fashion. Extensions are not limited by having to all fit into a single address or overlay space. Practically speaking, Frigate is limited by runtime resources (i.e., number of simultaneous running servers).

Frigate provides an object-oriented interface to the file system. This provides a single coherent client interface paradigm, rather than a hodgepodge of ad hoc interfaces. Variants are cleanly described through interface inheritance. The object interface also provides the ability to use polymorphism with file operations. Frigate's late binding mechanism allows implementation to be changed right up until a server is required. Since the actual implementation is not part of the client program, no recoding, recompilation or relinking is ever necessary to make full use of polymorphism.

The class provider is given a structured environment that merges vnode and ILU method programming into a common framework. Frigate remains compatible with Stackable Layers but with a better programming interface. The familiar file descriptor programming model is also available to program methods.

True encapsulation that cannot be bypassed, accidently or purposefully, is provided around file entities. Redefined vnode operations can enhance the behavior of old programs without any change to the program.

Frigate provides a mechanism for flexible version management. When new versions are installed, there is no need to reboot or restart the system. Multiple implementations can run side by side with no interference. New runtime instances are automatically assigned the updated versions. There is no need to hunt down every document instance to update it.

Except for initial installation of the framework, all facilities can be used with full capabilities by unprivileged users. Anyone can write, install, debug and use extensions to the file system.

5 Safety and Security

Frigate extensions are placed in server processes outside of the operating system address space. The process boundary around each server acts as a firewall. Buggy or malicious behavior is confined to the server process. Extensions communicate with the operating system through Stackable Layers transport layer interfaces and via system calls. Extensions do not have direct access to the operating system address space. Thus, the operating system is protected from the server just as well as it is protected from any other user process.

During the servicing of a method, the server holds the same access rights as the calling process. In between method calls, the access rights are changed to that of a special unprivileged Frigate user. The rights granted during method service are identical to that given to a program executed by a user. The difference is that access rights are being granted on a lower level of granularity and access rights change over the life of the server. It is not possible to use the varying credentials to accumulate access rights. Even if access is granted previously, the next access may be barred on the basis of the submitted credentials. Overall, malicious extension code cannot do anything not possible through ordinary malicious program execution.

However, Frigate does offer more danger as a Trojan horse. Precisely because Frigate can operate transparently, it is somewhat easier to unknowingly run an untrusted Frigate extension than to execute an untrusted program. This is especially true if a user defines a malicious vnode method. Frigate attempts to mitigate the problem in two ways. First, privileged (root) clients do not pass their rights to Frigate servers. Thus, the only actions that a Frigate server can perform for a root user are those that any non-privileged program could. Root users can always disable Frigate, if necessary. In addition, it is always possible for users to examine an accessible file entity for Frigate attributes. (Frigate does not allow the relevant vnode operations to be redefined.) In this way, a user can safely determine if a suspicious file entity is an untrusted Frigate document by examining its attributes.

The Frigate repository is shielded from unauthorized modification by file protections. Requests typically issued by the operating system are only accepted from protected ports. Only class owners can modify a document class and its implementations.

Frigate offers complete encapsulation of documents. Since Frigate is a part of the operating system and below the system call interface, it cannot be bypassed accidently or maliciously. Frigate allows any specified operation to be intercepted, except those used by Frigate itself. This ensures that security or data integrity mechanisms will not be bypassed. Malicious users cannot defeat encapsulation by altering the attributes on documents. Only the owner or the root user can change the attributes on a document.

6 Compatibility

Frigate remains compatible with the vnode programming model. As previously described, some Frigate extensions provide their new functionality exclusively through vnode methods invoked indirectly through the UNIX system call interface. As a result, this type of extension provides complete backward compatibility for existing programs. Old programs work "as is" and require no changes at all, not even recompilation or relinking, to take advantage of the new functionality. This preserves the current software investment. In many cases, this means that application programmers do not need to learn new paradigms to take advantage of Frigate capabilities.

Since Frigate is packaged as a Stackable Layer, Frigate also offers an inherent form of forward compatibility. The other layers of the stack may be updated or reconfigured independently of Frigate. This means that a Frigate system can take advantage of new features in other layers without any change to Frigate itself. For example, an enhanced storage substrate might be provided by substituting a new storage layer for the standard UFS layer.

Frigate also offers *incremental* use. While Frigate offers a new paradigm, one is not forced into an "all or nothing" decision to use it. Frigate can coexist with standard UNIX. A user need only use Frigate as much as is desired. There is no need to conform all activity to the new object-oriented environment. Porting of existing facilities is not necessary because the current environment continues to exist on a Frigate system. Frigate documents are distinguished by class and version attributes. Frigate documents may

be stored in any file system that can support those attributes. Otherwise, there is no restriction on storing ordinary files and Frigate documents in the same volume.

In some cases, it may desirable to port old applications to the object-oriented Frigate paradigm. For example, the benefits of being able to write generic clients, which take advantage of polymorphism, may justify the porting cost. In other cases, applications may wish to use Frigate's version management system. The incremental nature of Frigate aids application migration. Migration may be accomplished by "wrapping". Old files are converted to documents by wrapping class and version attributes around them. The addition of the Frigate attributes brings them under Frigate management. Old programs are "wrapped" as methods by having method handlers simply execute the programs directly. In this way, an old application is quickly enabled to run under Frigate. Further changes may, of course, require more extensive restructuring.

Frigate is also compatible with standard languages: OMG IDL and ANSI C. All inheritance support is confined to the ILU interface generator. Thus, no extensions to C are necessary and the investment in C compilers and programming environment can be preserved.

7 Performance

We measured Frigate's performance in two respects. First, we measured the cost of merely having, but not using, Frigate's framework. We also compared the performance of Frigate to alternative solutions.

All of our performance measurements were carried out under a SunOS 4.1.1 kernel on SPARC IPC (25 MHz, 15.8 MIPs) machines with 12 MB of memory and a SCSI 207 MB (3.5", 3600 RPM) disk. The operating system was modified to use Stackable Layers. All test results are derived from 30 runs.

7.1 Framework Overhead

We would like Frigate to have minimal performance impact when we are not using its facilities. To measure this impact, we ran the Modified Andrew Benchmark [24, 37] in a file volume with and without the Frigate service.

No direct use was made of Frigate in the benchmark. The configuration without Frigate was simply the standard UFS, packaged as a Stackable Layer. The configuration with Frigate included a Stackable Layer to implement extended (class and version) attributes and the actual Frigate Dispatcher layer. The

	Layer Configuration			
	UFS		Dispatcher Attribute UFS	
Phase	mean	stddev	mean	stddev
mkdir	4.3	2.28	4.8	1.04
cp	11.6	2.01	16.9	1.93
find	10.1	1.50	11.8	1.84
read	15.4	0.97	16.2	0.96
make	103.0	1.85	105.6	1.77
user	83.0	0.30	83.3	0.28
sys	37.7	0.35	39.9	0.36
elapsed	149.8	3.12	161.1	2.39

All runtimes in seconds.

Phase	Percentage
mkdir	112 %
cp	146 %
find	117 %
read	105 %
make	103 %
user	100 %
sys	106 %
elapsed	108 %

Frigate time as percent of UFS time.

Table 1: Non-Use Times

results for each configuration and each phase of the benchmark are shown in Table 1. Frigate times are also shown as a percentage of UFS runtime.

Overall, the overhead of having Frigate, but not using it, amounts to an 8% elapsed time penalty. Additional measurements were made with just the extended attribute layer to see how much of the overhead came from that support layer. About seven of the eight percent of overall overhead comes from the extended attribute support layer. For general use, Frigate's overhead would have to be reduced. The obvious strategy would be to improve the performance of the attribute service.

7.2 Comparison to Library

Our second performance study compared Frigate to a user-level library. In many cases, the unprivileged user has few alternatives but to implement his extension as some type of user-level library. Our example extension provided file encryption services. The encryption algorithm is an enhanced version of the one used in the German World War II Enigma

File Size (KB)	Library		Frigate	
	mean	stddev	mean	stddev
8	0.3	0.05	1.3	0.44
80	2.8	0.23	4.7	1.56
800	26.4	0.18	32.5	0.55
8000	295.6	5.80	343.2	6.58

All runtimes in seconds.

File Size (KB)	Percentage
8	433 %
80	168 %
800	123 %
8000	116 %

Frigate time as percent of Library time.

Table 2: Single Client Times

File Size (bytes)	Library		Frigate	
	mean	stddev	mean	stddev
1	0.2	0.05	1.0	0.23
8 K	5.5	0.22	1.7	0.26
80 K	52.4	0.97	8.4	0.51
800 K	556.3	109.77	74.1	0.20

All runtimes in seconds.

File Size (bytes)	Percentage
1	500 %
8 K	31 %
80 K	16 %
800 K	13 %

Frigate time as percent of Library time.

Table 3: Shared File Performance

encryption machine [16]. While not a strong encryption method, this example did provide a predictable processing overhead on each read and write.

The library implementation of our encryption extension simply followed any read of the encrypted file by a function call to decrypt the incoming block of data. Writes were preceded by a call to encrypt the outgoing data. File operations were performed on a UFS file system without any additional stackable layers.

The Frigate implementation redefined the read/write vnode method to provide transparent encryption and decryption. A new ILU method was added to provide the session key to the extension. Because Frigate is positioned below the system call interface, file data could be buffered and shared between clients in the clear. To prevent unauthorized parties from gaining access to clear text data, other vnode methods were also redefined with added security features.

Our first experiment used a single client, which provided a mix of file I/O operations and compute processing. Each block in the file was read, modified, and written back into another location in the file. The elapsed time of the library and Frigate implementations is shown in Table 2. Frigate times as a percentage of library runtimes are also shown.

Overall, we see for small cases (e.g. 8 KB file size) that the library performs much better. The reason for this result is the server startup time for Frigate. As file size (and hence overall I/O) increases, the server startup time is more effectively amortized over the entire run. In both library and Frigate cases, the same amount of processing is done by the driving client application and in encryption operations. The difference in performance comes from the lower throughput of Frigate. The additional overheads in Frigate include the two additional stackable layers, the limited speed of our transport, and the need to context switch to the server process to handle requests. For general use, further performance tuning of Frigate is probably necessary. In this case, the largest payoffs in improving performance are in improving our IPC throughput and server startup time.

Our second experiment involved two clients sharing access to an encrypted file. One process read, modified, and wrote each block of the file. Before proceeding to the next block, the second client also read, modified, and wrote the same block. The entire file was processed 10 times in this fashion.

The elapsed runtimes for the library and Frigate implementations are shown in Table 3. The runtimes for Frigate as a percentage of the library runtimes are also shown. The file size, which is directly tied to the total amount of I/O, was varied in our experiment. (The 8000K case was omitted because the library solution had an excessive runtime.)

The server startup cost again makes the library implementation faster in the smallest cases. Since so little data is actually transferred in the one byte file case, it essentially only measures the overhead of the framework. However, in the larger cases, Frigate does significantly better. The ability to share text in the clear reduces runtime substantially. In the largest case shown, Frigate runs in less than one-seventh the time of the library implementation. The savings is

mostly due to avoiding redundant encryption and decryption operations. Frigate need only perform these operations between its buffers and the disk. The library implementation must perform such operations on each I/O. This fact more than compensates for any I/O throughput reduction suffered by Frigate. In this case, Frigate's architecture provides a unique advantage not otherwise available.

8 Related Work

Frigate is comparable to a number of different systems. In general, we classify these systems according to their structure and functionality. A key question is where the extension is added to the overall system. File service extensions can be added to client processes, system libraries or the operating system. If extensions are implemented in servers, then some sort of "hook" is inserted into one of these locations to intercept the relevant calls.

Frigate uses servers. The intercept mechanism for Frigate is the Dispatcher layer, which intercepts selected vnode operations inside the operating system. Since all file accesses pass through the layer, Frigate can ensure that an extension has complete control over file operations. Frigate's user-level servers allow easy development. The process boundary around servers also protects the operating system from buggy extensions. The drawback is that performance can suffer due to the need for IPC between the Dispatcher and server. Frigate's object-oriented environment is provided for the client and server by using a CORBA based solution. The intercept mechanism in between is essentially ignorant of objects other than vnodes.

8.1 Languages & Libraries

Programs written in object-oriented languages can use objects with support for persistent storage. Typically, such objects can provide a high-level method interface appropriate to the application instead of the primitive system call interface. Support for persistence may be a builtin language feature. An elementary example is found in the Eiffel programming language [34]. The "storable" class provides methods to read and write an internal language representation of an object to a file. Other classes gain persistence by merely inheriting the storable class.

When languages have support for persistent objects, they can offer a natural, enhanced filing interface that is fully integrated with the rest of their language environment. On the other hand, their rich world is not available outside of their own language environments. They are not integrated with the operating system and cannot provide any encapsulation of persistent objects outside of the language environment. These systems and others that choose to remain above the system call interface can be free of the problems inherent to developing in the operating system address space. However, by not being able to go below the fixed system call interface, their flexibility and ability to enforce policy is limited. For example, it is difficult above the system call interface to even ensure unambiguous file identity. As we saw in the performance tests, it can also be a performance liability.

Most CORBA [36] based object frameworks are designed to integrate with target language environments. Thus, for the most part, they share the characteristics of language-based solutions. Object frameworks with "applet" and "serverlet" features offer a novel form of modularity. However, when actually executing, they too share the characteristics of language-based solutions. Some other object framework cases, such as OLE 2 [35] and ActiveX [8], are less clear, since it is difficult to determine how much has been subsumed by the operating system.

Another related approach uses libraries. Commonly used system libraries are modified to enhance the behavior of file operations. Often these libraries are shared and dynamically loaded. This allows existing compiled programs to use the modified library without recompilation. Of course, statically linked programs will not be able to take advantage of redefined, enhanced operations and will break encapsulation. Examples of library systems are the 3-D File System [29], COLA [30] and IFS [12]. These particular systems provide additional (albeit not object-oriented) functionality by intercepting library calls to the standard UNIX system calls. Like language-based solutions, library systems reside above the system call interface and thus inherit the associated tradeoffs.

8.2 Servers

Another form of operating system extension is the server. The server usually executes at user-level outside of the operating system (or at least outside of privileged execution modes). The intercept mechanism may be above or below the system call interface. Intercepted calls must be passed to the extension by some form of IPC. This may cause some loss in performance.

The Object-Oriented File System [47] intercepts UNIX system calls at the library level and passes them to servers using pipes or UNIX domain sockets. The modified library must be explicitly linked

with clients and thus will only work with new applications. Despite its name, it does not offer an object-oriented programming interface. There are no classes, methods, inheritance, etc. in this system.

The Distributed Object-Based System (DOBS) [11] provides automatically started servers that respond to object methods. Communication between client and server process is through Sun-RPC. This system is similar in flavor to the ILU part of Frigate. DOBS defines its own simple interface language to describe methods. It does not appear to have the ability to manipulate any non-file objects, nor is there any inheritance. Also, it does not have any integration with the UNIX file operations. No existing UNIX file operations can be redefined; one can only define new operations. As a result, the system cannot be used to affect the behavior of existing programs and encapsulation is not enforced.

Some operating systems offer special debugging mechanisms to intercept system calls. As long as the intercept can be properly set up, complete encapsulation is possible. Interposition Agents [26] use this strategy. An object-oriented toolkit is provided as a framework for implementing new services. However, the toolkit is not exposed to clients. Instead, each agent is provided with a symmetric system call interface allowing the layering of agents. Thus, clients continue to see the same unextended system call interface. In this system, agent(s) are attached to specific clients and are apparently not shared.

The File Monitor Interface [49] provides a user-level server facility that responds to vnode calls intercepted inside the operating system. The intercept mechanism is similar to the Frigate Dispatcher and can selectively intercept calls based on special file attributes. However, there is no provision for an object-oriented programming model.

Similar facilities were also built for other operating systems. Pseudo-File-Systems [51] were developed for Sprite [38]. Userfs [15] provides this service for LINUX. Stackable Layers also provides this functionality on vnode operations through transport layers. These three systems use the mount mechanism to provide a volume granularity redirection to servers. They also require servers to be started prior to use.

Watchdogs [4] and Extensible Streams [41] go somewhat further in that servers are started automatically, if none is running. Both also use type identifiers similar to Frigate's class attributes. This allows specifying servers on a file level granularity.

Micro-kernels take the server approach and apply it to reorganize the entire operating system. A modest number of basic abstractions are implemented in a small ("micro") kernel. The bulk of the operating system is moved into separate servers. Examples include Mach [1] and Chorus [45]. Exokernel [13] and Cache Kernel [9] attempt to push this organization as far as possible. In a micro-kernel, the file system is usually implemented by one or more servers. In the process of reorganization, many micro-kernels and their file systems have also acquired an object-oriented flavor. An example is Mach 3.0 [17, 46]. While this object orientation is of great use in the *internal* development and specialization of the operating system, it is not generally exported to users. Users continue to see the same unextended system call interface.

8.3 Extensible Operating Systems

A number of approaches to increase extensibility of the traditional file system have been tried. Early attempts to add extensibility interfaces to the operating system included streams [42] and dynamically loaded device drivers. The latter is included in systems such as SunOS and Chorus [2]. Other attempts to restructure the file system include VFS [28] and UCLA Stackable Layers [22, 23], which we have already described. An alternative model of stacking is described in [44]. The Spring operating system [27, 40] also offers stacking with additional object-oriented features.

Recently some new operating systems extend their functionality by downloading "safe" modules into the operating system. The modules are made safe by restricting address references [48] or by using safe languages. An example of the latter is the SPIN operating system [5], which uses Modula-3 [7] as its extension language. With the use of the techniques mentioned above relative safety can be assured. In some cases, though, there is limited space to add extension code and thus large extensions are not possible.

Unfortunately, any facility requiring development and debugging in the kernel address space needs special tools and privilege. Also, these tools were designed for expert specialization of the operating system. They are not meant for casual users. However, these approaches potentially perform better than servers, because there is no need to cross process boundaries to reach the extension.

8.4 Object-Oriented Operating Systems

Another approach is to provide operating systems that are built on an object-oriented paradigm from the ground up. The services they offer, including

the file system, are object-oriented. Extensibility is provided by using subclasses and polymorphism. This is to be distinguished from merely writing a system in an object-oriented language. See [21] for further discussion on this point. Of course, the two are not mutually exclusive; many such systems are actually written in an object-oriented language or use one for its client interface.

Choices [6, 32, 33] implements an object-oriented operating system written in C++ with some extensions. Operating system abstractions, including files, are provided as objects in a class hierarchy. Extensions can be added by subclassing existing classes. However, the extension framework is not targeted for use by the ordinary user. Rather, it is oriented toward controlled specialization of the operating system by privileged expert users. Choices does try to address the problems of operating system debugging by providing a user-level emulator.

Clouds [10, 39] offers a complete object-oriented operating system. All services in Clouds are offered as objects accessed through capabilities. The radical persistent object model makes no distinction between memory objects and file system. Essentially, all objects are located in a large distributed virtual memory. Objects in Clouds are relatively large grain with operating system enforced encapsulation. The COOL [20] system provides a similar model built on top of the Chorus micro-kernel.

True object-oriented operating systems can provide powerful extensible environments. Their main drawback is their incompatibility with the rest of the world. The current investment in software is lost. Any desired feature must be ported and possibly rewritten to fit with the new paradigm. One must make an "all or nothing" switch to the new system. Only a few systems such as Solaris MC [3] and Frigate explicitly attempt to integrate compatible use with a new object-oriented operating system interface.

9 Future Work

Frigate presents a number of directions for future work. These concepts are enhancements that can be incrementally added to the current, fully operational system.

Frigate is currently implemented on the SunOS 4.1.1 infrastructure of Stackable Layers. As Stackable Layers is ported to other platforms, Frigate can also be ported with only modest effort. On its current platform, the Frigate client environment could also be expanded to include other ILU languages such as C++ and Modula-3 [7].

Some improvements could also be made to the server structure. The most fundamental would be to allow object types to be dynamically loadable. This would allow implementations to be decomposed into a shared server executable and an application specific set of dynamically loadable implementations. Servers would tailor themselves while running and load implementations only according to demand. Currently, a compound document server must have implementations of all possible component classes linked into the executable. In a dynamic architecture, compound documents would load only what was demanded by the access pattern of their constituent components. New classes and versions would not require relinking a server executable. In effect, we would be extending late binding to server binary construction as well.

Another possible improvement is to allow servers to be distributed on other hosts. This would allow additional server configuration flexibility and better load balancing. To allow this possibility, a new distributed alternative to the Stackable Layers user-to-kernel transport must be provided along with some enhancements to the binding process.

A number of performance improvements are possible as well. Probably, the largest payoffs in performance are in improving the separate attribute service, the communication throughput with the server and the server startup time. For servers on the same host, considerably better performance is probably possible. Current kernel-to-server transports are based on implementations of NFS. Much more efficient interprocess communication could be used. Improvements would probably use strategies similar to those used in micro-kernel designs (see [31]).

Server startup is an expensive process involving new process creation and process image overlay. When in the critical path of execution, this can result in a substantial delay. One strategy to improve things would be to cache servers. Servers would not be killed until a timeout period had expired. Further uses of the same implementation would not require a new server to start. Another complementary strategy would start up a number of dynamically loadable servers ahead of time. Unassigned servers may even be preloaded with some implementations. As servers are needed, they are assigned from the pool of servers. In this way, the server startup time is avoided, though, the time cost for dynamic loading must still be paid.

10 Conclusion

Operating systems and the file systems they contain have been designed as general purpose instruments. Specialized file system support for applications could provide great benefits. Unfortunately, it is very difficult to extend operating systems and file systems. What is needed is a file system infrastructure with extensibility designed in. While some extensibility features have appeared, they do not go far enough in providing open access for extensibility.

Frigate attempts to provide such open access to all users, not just to those with system privilege or specialized training. The philosophy driving Frigate is that the user or ordinary programmer is best equipped to add value to his applications, not some distant expert. Therefore, the challenge is to expose the power of underlying mechanisms in a safe and easy to use way.

Towards this end, Frigate combines a user-level server framework, which is fully integrated with the operating system, with an object-oriented interface. The user-level structure provides adequate performance and, at the same time, frees developers from the constraints of development in the operating system address space. It allows the freedom to provide an implementation that does not have to mirror the structure of a corresponding implementation inside the operating system address space. It is *expected* in this environment that untrusted extensions will be run. We use the server process boundary as a firewall, protecting the operating system and other extensions.

The object-oriented framework provides a coherent programming model for extension that provides the benefits of inheritance, polymorphism and encapsulation. Users do not have to learn a new programming paradigm for each extension. Inheritance allows the leveraging of previous work. Polymorphism allows generic programs to be written that need not change as new versions or types appear. Frigate also provides full encapsulation that is enforced and cannot be bypassed.

Frigate provides compatibility. By being able to redefine vnode operations, Frigate can extend the behavior of existing programs. Vnode operations are smoothly integrated into the object-oriented programming model. Frigate uses standard languages and compilers. The implementation of object-oriented mechanisms are carefully isolated so that current compilers need not be disturbed. The value of the current software investment is preserved. By using Stackable Layers technology, Frigate can also take advantage of new work packaged as layers.

Frigate allows incremental use. Frigate objects can be freely mixed in the same file system with ordinary files. Frigate code can cooperate with other programs using the standard system call interface. Frigate does not force one to chose between paradigms. Frigate can be used as much or as little as desired. It is not an "all or nothing" approach.

Frigate also provides a structure for controlled change without any disruption of operation. Not only are interfaces named but versions are as well. Particular versions can be named and relevant distinctions can be maintained. Yet at the same time, versions can be transparently upgraded. The installation and upgrade of versions never causes a need to reboot, reinitialize or interrupt operation of the system.

Frigate scales well. Interfaces and versions can be developed independently of each other without fear of name clashes. The namespace for interfaces and versions is assigned in a distributed fashion without the need for a central registry. Practically speaking, Frigate is only limited by runtime resources for server processes and not by arbitrary design limits.

Other systems have only addressed a few of these aspects. In short, we believe Frigate uniquely opens the file system to the world.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer Conference Proceedings 1986*, pages 93–112. USENIX Association, June 1986.

[2] F. Armand. Give a Process to Your Drivers! In *Proceedings of the EurOpen Autumn 1991 Conference*, Sept. 1991.

[3] J. M. Bernabeu-Auban, V. Matena, and Y. A. Khalidi. Extending a Traditional OS Using Object-Oriented Techniques. In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems*, pages 53–63. USENIX Association, June 1996.

[4] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. In *USENIX Winter Conference 1988 Proceedings*, pages 267–275. USENIX Association, Feb. 1988.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Backer, C. Chambers, and S. Eggers. Extensbility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284. ACM, Dec. 1995.

[6] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-

Oriented System in C++. *Communications of the ACM*, 36(9):117–126, Sept. 1993.

[7] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report (Revised). Research Report 52, DEC Systems Research Center, Nov. 1989.

[8] D. Chappell. *Understanding ActiveX and OLE.* Microsoft Press, Redmond, WA, 1996.

[9] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, Nov. 1994.

[10] P. Dasgupta, R. J. LeBlank Jr., and W. F. Appelbe. The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work. In *The 8th International Conference on Distributed Computing Systems Proceedings*, pages 2–9. IEEE Computer Society Press, June 1988.

[11] P. Dewan and E. Vasilik. Supporting Objects in a Conventional Operating System. In *Proceedings of the Winter 1989 USENIX Conference*, pages 273–285. USENIX Association, Jan. 1989.

[12] P. R. Eggert and D. S. Parker. File Systems in User Space. In *Proceedings of the Winter 1993 USENIX Conference*, pages 229–240. USENIX Association, Jan. 1993.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266. ACM, Dec. 1995.

[14] J. Feiler and A. Meadow. *Essential OpenDoc.* Addison-Wesley, New York, NY, 1996.

[15] J. Fitzhardinge. *Userfs – Filesystems Implemented as User Processes.* Softway Pty. Ltd., Aug. 1995.

[16] S. Garfinkel and G. Spafford. *Practical UNIX Security.* O'Reilly & Associates, Sebastopol, CA, 1991.

[17] P. Guedes and D. P. Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. In *1991 International Workshop on Object Orientation in Operating Systems Proceedings*, pages 114–117. IEEE Computer Society Press, Oct. 1991.

[18] R. G. Guy. Ficus: A Very Large Scale Reliable Distributed File System. Technical Report CSD-910018, UCLA Computer Science Department, June 1991.

[19] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–71. USENIX Association, June 1990.

[20] S. Habert and L. Mosseri. COOL: Kernel Support for Object-Oriented Environments. In *OOPSLA/ECOOP '90 Proceedings*, pages 269–277. ACM, Oct. 1990.

[21] G. Hamilton, Y. A. Khalidi, and M. N. Nelson. Why Object Oriented Operating Systems are Boring. In *1991 International Workshop on Object Orientation in Operating Systems Proceedings*, pages 118–119. IEEE Computer Society Press, Oct. 1991.

[22] J. S. Heidemann. Stackable Design of File Systems. Technical Report CSD-950032, UCLA Computer Science Department, Sept. 1995.

[23] J. S. Heidemann and G. J. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.

[24] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[25] B. Janssen, D. Severson, and M. Spreitzer. *ILU 1.8 Reference Manual.* XEROX Palo Alto Research Center, Mar. 1995.

[26] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93. ACM, Dec. 1993.

[27] Y. A. Khalidi and M. N. Nelson. Extensible File Systems in Spring. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14. ACM, Dec. 1993.

[28] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer Conference Proceedings 1986*, pages 238–247. USENIX Association, June 1986.

[29] D. G. Korn and E. Krell. The 3-D File System. In *Proceedings of the Summer 1989 USENIX Conference*, pages 147–156. USENIX Association, June 1989.

[30] E. Krell and B. Krishnamurthy. COLA: Customized Overlaying. In *Proceedings of the Winter 1992 USENIX Conference*, pages 3–7. USENIX Association, Jan. 1992.

[31] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–188. ACM, Dec. 1993.

[32] P. Madany, R. Campbell, V. Russo, and D. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In *ECOOP '89 Proceedings.* Cambridge University Press, July 1989.

[33] P. W. Madany, D. E. Leyens, V. F. Russo, and R. H. Campbell. A C++ Class Hierarchy for Building

UNIX-Like File Systems. In *USENIX C++ Conference 1988 Proceedings*, pages 65–79. USENIX Association, Oct. 1988.

[34] B. Meyer. *Object Oriented Software Construction*. Prentice-Hall, New York, NY, 1988.

[35] Microsoft Corporation. *Object Linking & Embedding Version 2.0 Design Specificaton*, Apr. 1993.

[36] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report 96.08.04 (Revision 2.0), Object Management Group, July 1996.

[37] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256. USENIX Association, June 1990.

[38] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[39] D. V. Pitts and P. Dasgupta. Object Memory and Storage Management in the Clouds Kernel. In *The 8th International Conference on Distributed Computing Systems Proceedings*, pages 10–17. IEEE Computer Society Press, June 1988.

[40] S. Radia, P. Madany, and M. L. Powell. Persistence in the Spring System. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 12–23. IEEE Computer Society Press, Dec. 1993.

[41] J. Rees, P. H. Levine, N. Mishkin, and P. J. Leach. An Extensible I/O System. In *USENIX Summer Conference Proceedings 1986*, pages 114–125. USENIX Association, June 1986.

[42] D. M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, Oct. 1984.

[43] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[44] D. S. H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the Summer 1990 USENIX Conference*, pages 107–117. USENIX Association, June 1990.

[45] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Technical Report CS/TR-90-25, Chorus systèmes, Apr. 1990.

[46] J. M. Stevenson and D. P. Julin. Mach-US: UNIX on Generic OS Object Servers. In *Proceedings of the 1995 USENIX Technical Conference*, pages 119–130. USENIX Association, Jan. 1995.

[47] S. Summit. Filesystem Daemons as a Unifying Mechanism for Network Information Access. In *Proceedings of the Winter 1994 USENIX Conference*, pages 63–77. USENIX Association, Jan. 1994.

[48] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, Dec. 1993.

[49] N. Webber. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the Winter 1993 USENIX Conference*, pages 219–228. USENIX Association, Jan. 1993.

[50] J. Weidner. An General Purpose Extended File Attribute Service as a File System Layer. Technical report, UCLA Computer Science Department, 1997. To appear.

[51] B. B. Welch and J. K. Ousterhout. Pseudo-File-Systems. Technical Report CSD-89-499, University of California, BerkeleyComputer Science Division, Oct. 1989.

[52] L. Zahn, T. H. Dineen, P. J. Leach, E. A. Martin, N. W. Mishkin, J. N. Pato, and G. L. Wyant. *Network Computing Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

Embedded Programming with C++

Stephen Williams, *Picture Elements, Inc.* steve@picturel.com

May 5, 1997

Abstract

This paper presents uCR, a C++ runtime package for embedded program development. We make the case that in certain situations embedded programming is best done without the aid of a conventional operating system. A programming environment in the form of a C++ runtime is presented, and the environment, including the C++ language, is evaluated for appropriateness. Important factors are code size, performance, simplicity and applicability to a wide range of embedded targets.

1 The Problem

It is common, when building a newly designed board, to install only a few components at a time and test the partially built board to protect expensive components, to validate portions of a design, or just to contain the hardware debugging problems. The first time power is applied to a board, often only the CPU, memory and ROM socket are installed. Naturally, software is usually required and a development environment that works in this case is necessary, especially as the board design and construction progresses.

Even complex designs can have real estate constraints, leaving no room for the extra hardware to support a full operating system. A case example of this is shown in Figure 1. In order to fit this design on a PCI card, extra parts like UARTS had to be left out, and program memory had to be kept to one flash and 2 DRAM chips.

Conventional operating systems usually serve two interesting roles: they abstract the target hardware, and they provide a means of loading and executing programs, often in separate protection domains. An operating system provides an operating environment, including but not limited to a device driver interface and a common interaction with the user. It is separated from applications by a kernel structure, bounded by trap handlers or some form of call gate that allows the operating system to function to some degree independent of and protected from the applications that it carries.

Several commercial embedded operating systems are available that run on the relatively conventional CPU in Figure 1, but most commercial operating systems, available in binary form, require board support packages written to provide the necessary support for the O/S, including a console, time ticks, and memory setup.

The ISE board (Figure 1) in particular has no serial port, so program loading must be done either by programming the socketed FLASH memory with a prom programmer, or writing into the board support package a console driver that uses the PCI bus to communicate as a console. The MON960 monitor [8] supports the latter, and the Cyclone-911 board [4] in particular can be used this way, given the appropriate host software.[1]

Although it is sometimes nice to have an operating system that is portable, and essential that certain libraries be portable, it is rare that an embedded program is, or should be, portable. The whole point of a program is to manipulate the specific toaster. There is no value being able to run the toaster program on the VCR. It therefore is rarely useful to have a device-driver interface in an embedded kernel—such can actually make things harder.

We questioned the prudence of forcing a kernelized operating system onto a board with only a few LEDS and an oscilliscope for debug output, and a ROM socket for input. We anticipated this happening often, as designing and building boards is our business. We also noted that the device driver interface of a kernel is pointless, and our targets typically run a single trusted program from reset to power off. We eventually concluded that we didn't really need an operating system at all.

This, then, became the chosen path. We wrote a minimal runtime to support C and C++ that works on the sorts of target boards expected, and we provided that support for a specific compiler, the GNU

[1] Picture Elements supplies with the ISE board a bootstrap loader that loads COFF files from the PCI bus. The loader is written using uCR and the techniques described in this paper.

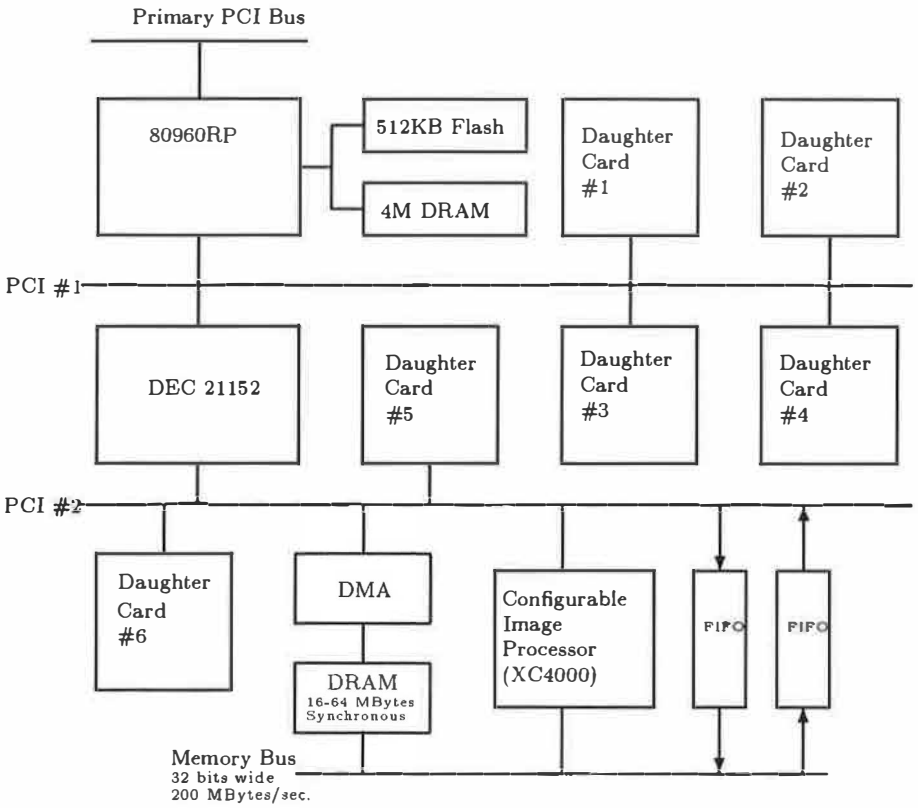


Figure 1: Imaging Subsystem Engine (ISE) Block Diagram [9]

GCC compiler. Writing the support for the compiler alone, we reasoned, would be easier then writing a board support package for compiler *and* operating system and would get everything needed without the added constraints of an operating system.

This runtime support for the compiler, called uCR[2], proved lightweight and powerful enough that we not only used it as the regular development environment, we used it to build bootstrap loaders and other programs in support of embedded development itself.

2 uCR, C and C++

The difficulty of porting embedded operating systems is more often dealing with the board, and not the CPU proper. It is the memory layout and I/O devices that make portable operating systems difficult. If one can reduce the development environment to something independent of the board, then there is only the CPU to worry about and not all the devices around it. uCR/i960 can run on any i960 without porting.[3]

Eliminate the devices from the environment, and eliminate the system call gate, and what remains is a runtime that connects the compiler to the CPU. In the embedded world, anything is possible and imposing irrelevent requirements like a console and a clock ticker can make things harder.

The problem, then, simply reduces to how one maps C++ to a CPU, with nothing else. Although board specific code still needs to be written, specifically the reset handler and application code, uCR makes no requirements other then those needed to support the compiler. This is a task to remind one about the nature of a programming language.

This is not quite the same as more conventional development environments requiring board support packages. Although uCR in practice needs board-specific startup code, it does not impose a style of interaction with the board and programmer. Typical systems require of the board support package console drivers, timer drivers (with the timer configured to tick at a specific rate) and package drivers to initialize the options you choose to include. uCR imposes no such requirements.

[2] uCR is an abbreviation of "Micro-C/C++ Runtime," pronounced U-C-R

[3] In practice, the uCR package, though not the core library, includes code specific to select boards in order to get the developer started writing more complex programs.

2.1 What uCR is

uCR is more properly called a C/C++ runtime than an operating system. A programmer writes an application program in C++ and compiles with the uCR headers. The compiler generates assembly code from the source files that the programmer writes, and what the compiler cannot do it delegates to the execution environment. uCR provides the execution environment for the generated code.

The programmer links the resulting object code with uCR libraries that fill in the parts left out by the compiler, and gets an executable image. This image is loaded into the target by prom programmer, ROM emulator, serial download—whatever works for you—and is executed.

uCR libraries add support for thread programming and interrupt handlers. These are features dependent on the CPU and not the board, so including them does not introduce board support problems. The uCR distribution also includes ancillary libraries that contain device classes, and other code that may not be specific to the CPU but is commonly used.

Interrupt handler support is also included with the uCR core library, because again it is a matter for the CPU and compiler how interrupt service routines are entered and left, and not specific to target boards. What the target boards do fix is the assignment of devices to specific interrupts, so uCR makes no attempt to guess such things.

2.2 What uCR is not

uCR is not a kernel, or a micro-kernel. There are no system calls and there are no task structures such as page translation tables or system call gates. Calls to uCR operations are ordinary function (or method) calls.

uCR also is not intended to abstract the board away. Operating systems that try to abstract the hardware away wind up instead requiring that the hardware be a certain way. That is frequently counter-productive. The uCR core does nothing to devices other than the CPU, and does nothing to get in between devices and the programmer.

2.3 Requirements of the Languages

Both C and C++ place some minimum requirements on the execution environment, but many of the constraints are imposed by the compiler, not the language. If some language construct can be handled easily by the CPU, then the compiler typically just generates the assembly code to deal with it. That is

what compilers are for. However, when something is too hard, it gives up and generates a call to external code.

The C language is relatively easy to compile and the compiler only generates call instructions for calls to external functions. Floating point emulation is often placed in a library as well, if the target can reasonably be expected not to have a floating point unit. The C++ language is a bit more interesting. It has difficult constructs that compilers often give up on, like dynamic memory allocation.

The C language standard has a substandard, the *freestanding* C standard [1], to guide the implementer on what can be left out of the development environement and still be worthy of the name "C". In a nutshell, libraries are optional in a freestanding environment. It is rare for an environment to not include some of the more important optional parts, though.

The C++ Working Paper has a similar substandard.[4] [3] The standard libraries are not required of a freestanding C++ environemnt. Only a few support libraries, some specific C library routines, and support for the "`new`" and "`delete`" operators are expected. Things like streamio are certainly not required of a freestanding C++ execution environment, although a specific implementation may choose to provide it.

Static initializers must obviously be done correctly. To fail to initialize static objects is a clear and gross error, but the compiler certainly does not know how to arrange that on my toaster CPU. That, like the minimum library support, becomes a matter for the runtime, namely uCR.

The g++ compiler generates external calls for new and delete. This way, memory allocation can easily be provided with some help at link time. Most targets have memory available for allocation, and some have several different kinds of memory for allocation. Even if the default allocation operators do not apply, the placement "`new`" operator has some interesting advantages.

2.4 Requirements of Common Sense

Ultimately, it is not enough to just make the compiler happy. The programmer using the compiler is the real customer and the programmer wants to make devices do interesting things. It is therefore not useful to have a beautiful and fast string manipulation library if the programmer cannot fit the program in memory.

[4] Strictly speaking, C++ doesn't yet have any standard at all.

Therefore, any practical system should make as much of the standard libraries as reasonable available to the programmer, without imposing extra costs. One programmer may not wish to pay for stdio, but another may find it worth while.

Finally, we wanted a lot of power out of uCR, but simplicity and efficiency were most important. It is intended to be a language runtime, so certain standards, such as POSIX [5], were not considered desireable for embedded applications. We also tried to keep the size and complexity of the programming interface small and understandable. When testing a new board design, or debugging an older malfunctioning board, simple and obvious software behavior has its own special value.

3 Object Oriented Design

By using C++, Picture Elements gained a chance to use object oriented techniques in an embedded context, with threads of execution and interrupts. The obvious potential object classes are:

- Threads,
- Synchronization variables,
- Devices,
- The Debugger,
- Various containers.

The various containers include ring buffers, lists, strings, and the other sorts of things one expects of object oriented designs, are not unique to embedded programming so will not be discussed here.[3, 10, 12]

Incidentally, the requirement of keeping uCR simple and to the point precluded creation of a large and complex system. Instead of a rich texture of objects and classes, we finished with a simple and elegant design, with a few general but very useful classes. A welcome benefit of this is that programmers can learn and be productive with uCR relatively quickly.

3.1 Threads as Objects

The thread programming interface for uCR was revised several times before the current interface was settled on. At first, we designed threads as objects with interesting methods and put them in ThreadQueue containers. Eventually, however, we chose to attach most of the thread methods to the `ThreadQueue` class and left the `THREAD` a passive, opaque object. The run queue, we reasoned, would

be a **ThreadQueue** that the programmer can manipulate like any other **ThreadQueue**. Threads in the run queue would be subject to execution, and could be suspended simply by pulling the thread from the run queue and placing it elsewhere.

```
class ThreadQueue {
    public:
        // Pass true to the flag to
        // put the thread in front.
      void enqueue(THREAD*, bool=false);
      THREAD*pull();
      THREAD*peek();
};
```

This proved successful, though occasionally cumbersome and less clear then the more conventional POSIX-style thread functions. uCR internally uses the **ThreadQueue** class for the run queue and suspension lists in synchronization primitives. These instances are generally hidden from the application programmers, who have ultimately chosen to use POSIX-style thread functions provided in the uCR library. Programmers may use the **ThreadQueue** class to implement new synchronization primitives, if desired.

The threads themselves are opaque objects of type **THREAD**, and are passed around to the **ThreadQueue** objects and thread manipulation functions like a thread identifier in a more conventional thread package. The **THREAD** object is a completely opaque token used by the programmer, and uCR, to represent the object that is a thread. This is more an abstract data type design then an object oriented design. It is a semantic quirk that this C abstract data type is much like a concrete object class in C++.

The idea of a thread as an abstract class with a virtual method for its behavior is known to us. Some call this paradigm an *active object*. [2] Active objects are different from passive objects in that they have their own thread of execution and activate passive objects by calling methods. Threads and interrupt handlers are two different kinds of active objects, synchronization primitives and devices examples of passive objects.

We experimented with active objects, but ultimately decided to implement threads using the **THREAD** token and a set of conventional thread manipulation functions. The traditional thread functions are well established, easy to use, small and generally don't come into play once the threads are created and started. However, the specific interface built-in to uCR allows for efficient implementation of active objects if desired.

3.2 Memory Heaps

Support for memory allocation in uCR is itself written in C++. However, special care must be taken that all the data structures for managing the default heap in particular be in place before any static initializers are called. To arrange for this, the default heap initializer is called separately and ahead of initializers by the startup function of uCR.

The **HEAP_SPACE** object is an object token like the **THREAD** type, and represents a segment of heap space from which memory may be allocated. The uCR startup creates an initial **HEAP_SPACE** object that is used by malloc and non-placement new operators.

Embedded systems often have different kinds of memory, for example SRAM for small private objects, or perhaps synchronous DRAM for manipulation of large images, and so uCR allows programmers to create other heaps in specified sections of address space.

The uCR support for memory allocation adds a placement new operator that takes as a parameter the **HEAP_SPACE** to use. Because of the way the heap data structures are designed, deallocation does not require a reference to the **HEAP_SPACE** object that created it. This feature was specifically included to support the delete operator. Any memory allocated with "**new**" or "**new(HEAP_SPACE*)**" can be deleted with "**delete**". This is necessary because "**delete**" cannot be overloaded to take a **HEAP_SPACE** parameter, and to require such would render "**delete**" unuseable for memory allocated from alternate heaps.

3.3 Synchronization Objects

These classes were obvious and successful. uCR includes several synchronization classes, most derived from the base class ISync. The **ISync** class is a concrete class that allows threads to wait for a general condition to become true, and allows other threads to notify of a possible change in state. This base class is the most primitive and general synchronization that allows threads to interact with other threads and interrupt service routines.[5]

```
typedef bool (*sfun)(volatile void*);
class ISync {
    public:
      void sleep(sfun fun,
                 volatile void*);
      void wakeup();
};
```

[5] ISRs may not block so may not wait for a condition, but can and usually do report a potential change in state.

The `ISema` class is a counting semaphore implemented with the `ISync` class. The only operations supported are increment and decrement (and initialize with a specific value). The methods are easily implemented with the sleep and wakeup operations of the `ISync` class.

The `ILock` class is a binary semaphore. In principle, it could be implemented with the `ISema` class, but it works out more efficiently derived directly from `ISync`. Operations for ILock are get and put, and do the obvious things.

`ISema`, `ILock` and other classes are implemented by deriving from the `ISync` class. They all have similar fundamental behavior that can be easily factored into the base `ISync` class. The `Mutex` class implements *monitor* style synchronization and does not fit well in the `ISync` class hierarchy, so it is implemented separately.

The `Mutex` class has "`enter()`" and "`leave()`" methods to enter critical sections of code. Only one thread may be active in a critical section, hence the synchronization. The `Condition` class is a way for a thread to sleep within a critical section. A thread sleeping on a condition is not considered active in the critical section so other threads can enter. However, the implementations of Mutex and Condition assure that at all times there is no more then one thread executing in the critical section.

3.4 Devices as Objects

uCR proper does not operate devices, or expect any to be present, but ancillary libraries include classes that drive various devices. The application may add more device classes simply by writing the code needed to manipulate device registers, and putting that code into classes.

Devices make good objects. Programmers seem to understand and respond well to this technique. As an example, a uCR library includes classes for communicating with a host through a PCI bus. The class "`BUS`" has a subtype `BUS::Device` that is the abstract type of a bus interface device. The `PLX9060` class is derived from `BUS::Device` and drives the PLX Technology PCI9060 [11] interface chip. The `I960RP` class is also derived from `BUS::Device` and drives the ATU function unit of the Intel i960rp [6] microprocessor.

The `BUS` class in Figure 2 uses `BUS::Device` objects to implement a channel protocol with the host processor. The abstract `BUS::Device` class has a minimum set of methods used by the `BUS` class for sending packets to the host and getting packets back.

The classes derived from `BUS::Device` implement the minimum methods (which are pure virtual) and others that the device can support in addition to the minimum requirements. This is a fairly classic object oriented design. A similar hierarchy exists for timers (in case you choose to use them) where the abstract `Ticker` class provides a common interface for a generic clock and the derived classes implement the virtual methods as necessary for the specific hardware available.

The `Ticker` class hierarchy in Figure 3 is another example of an object oriented device driver design, also taken from the uCR libraries. The `Ticker` base class provides the methods of a generic interval timer, that portable code may use. The concrete class derived from the `Ticker` drives the real hardware timer to implement the behavior of the abstract class.

Even when inheritance does not make sense, device driver code fits well into classes. For example, the `XC4000`[6] class has the method `device_configure()` for programming the device, but does not specifically support or require any derivation.

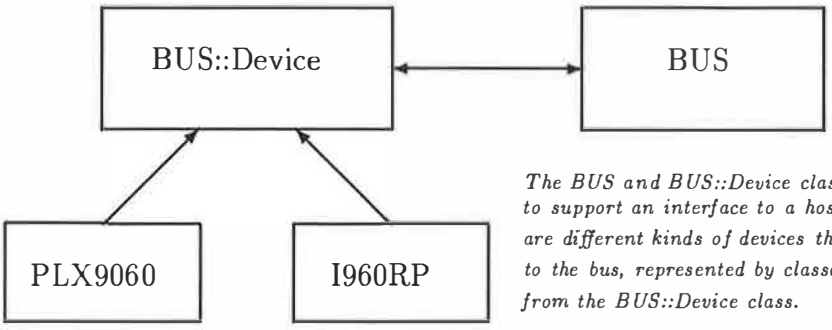
Device classes can be templates, too. The uCR libraries include a template class `LED` that is a driver for light-emitting diodes. This template is also useful for controlling general purpose output bits, and other miscellaneous jobs.

```
template <class RT> class LED {
    public:
      explicit LED(volatile RT*base);
      void set(unsigned idx);
      void clr(unsigned idx);
      [...]
};
```

Often, LEDS are connected to a register somewhere in the address space of the processor. The register may be a word, or a byte, or whatever the hardware costs and design allow. The programmer uses as the template parameter the integer type needed to access the word (for example "`char`" or "`unsigned long`") and passes to the constructor the address of the register. Individual bits of the register can then be set or cleared with the "`set()`" and "`clr()`" methods.

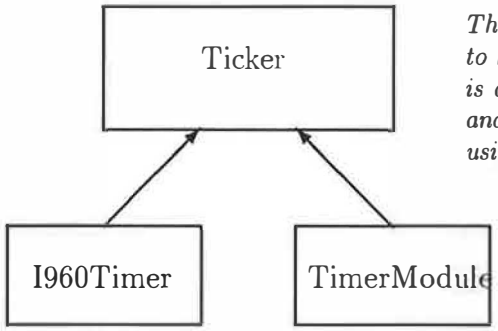
The nice feature of this template is that the programmer can use custom types for `RT` that for example store its value in memory outside the address space, such as I/O space.

[6] A Xilinx Field Programmable Gate Array



The BUS and BUS::Device classes cooperate to support an interface to a host bus. There are different kinds of devices that interface to the bus, represented by classes derived from the BUS::Device class.

Figure 2: BUS Class hierarchy



The Ticker is a general interface to hardware timers. The I960Timer is a Ticker using the i960Jx timer, and the TimerModule is a Ticker using the mc68322 timer module.

Figure 3: Ticker Class hierarchy

3.5 The Debugger

Some targets have sufficient I/O capabilities to support an interface to a debugger, so uCR provides the `GDB` class. A `GDB` object is an interface to a suitably modified version of the GNU Debugger GDB that runs on a host computer.

The `GDB` class is a concrete class that takes in its constructor a pointer to a suitable device class for use as the communication channel with the host. Once created, the debugger does not become available to the host until its "`go()`" method is called, generally by a special thread. This method never returns and forever communicates with the host, receiving and processing requests for action, memory, etc.

The uCR core leaves things like breakpoint traps and faults available to the programmer, so the `GDB` class uses standard interfaces to gain access to threads and the faults they make. The debugger runs in a thread of its own, so can be said to be an active object. As an active object, it is free to manipulate other threads, stop them, run them, examine memory, etc.

It turns out that not much of what the proxy needs to do is CPU specific, and other then actual I/O that communicates with the hosts, none of it involves board details. Many of the details of the target CPU are managed by the host GBD program, leaving only some CPU specific cache management and fault handling for the `GDB` class to cope with.

3.6 Summary

Table 1 summarizes the core set of types, plus a few others. Notice that this list is very short indeed. The uCR core really exists to provide a runtime context for the C++ program, and not to provide a lot of features. However, the thread and heap support tends to be compiler and CPU specific so must be provided in the core.

Device drivers are not required by the compiler, or by uCR, but some types of devices are common enough to offer in a packaged library some classes to help the programmer. The `BUS`, `Ticker` and `LED` classes are examples of class library support for devices. The list of device types here is not exhaustive. By putting device support in a library instead of a kernel, the drivers can be brought in by the linker automatically if the device is used by a program.

Type	Description
`THREAD`	Opaque object representing a single thread
`ThreadQueue`	Container for THREAD objects
`ISync`	Interrupt safe synchronization variable
`ILock`	binary semaphore
`ISema`	counting semaphore
`Mutex`	MONITOR style thread synchronization
`Condition`	Condition variables used with Mutex objects
`HEAP_SPACE`	Opaque object representing a memory heap
`BUS`	BUS interface abstract class
`Ticker`	Timer device abstract class
`LED`	L-E-D and output bit driver template
`GDB`	Proxy for the GNU Debugger

Table 1: Common uCR Object Types

4 Performance

All the cleverness in the world is useless if the resulting design performs poorly. This in fact is where we met the most resistence from C programmers. The premise of most criticism is that C++ code leads to inferior executables, either bloated by support for C++ capabilities, or somehow merely unoptimizable.

Much effort went into proving by example that tight and efficient C++ code is certainly possible. Since we chose to stick with a specific compiler, we had the luxury of studying the output assembly code and working it until the assembly couldn't be further improved. That experience has led to some generalizations.

4.1 Branching and Method Calls

Compilers are good enough that sequential sections of code are optimized very well and branches are the bulk of the execution time and code space. These are the logic of the program, and cannot usually be eliminated. The optimizer can be helped by avoiding conditional code, short loops should be unrolled, and it may be best to not branch around useless code in certain cases.

Object oriented designs, and C++ programs in particular, tend to introduce many smaller functions that perform near-trivial operations. For example:

```
class Foo {
        [...]
      int value()
          { return value_; }
      unsigned size() const
          { return 16; }
        [...]
};
```

The call to `value()` can be reduced to a single "`mov`" or "`ld`" instruction on most types of CPUs, and the `size()` method can be optimized completely away. However, if those methods were not defined inline then the compiler would be forced to generate a call and a return, would need to invalidate registers and/or shift register windows, and otherwise multiply the complixity.

By inlining, the call instruction is eliminated and the basic block is expanded to surround the method invocation. Subexpression elimination and register allocation can be applied more globally and code around the method call shrinks along with the method call.

C programs can also benefit from this technique. Linux source code, for example, is filled with tiny inlined functions of this sort. They are easier to read then similar macros, and more clearly express intent to the compiler.

4.2 Virtual Methods

Implementing virtual methods usually means an extra memory access before the call to the method. This sounds like a performance problem, but in fact it turns out to be a useful optimization, when used wisely.

A call to a C++ virtual method is often used to

invoke a context-specific behavior. For example, one might use virtual methods to perform I/O on an abstract class Device. The typical C equivilent is to keep function pointers in a function, and use those function pointers to perform the operation.

The typical C code has the function pointers in the structure with the other variables, making the structure larger. Every instance has pointers to all the functions, so there are many pointers to every function. A more efficient way to do this is to keep in the structure only a pointer to a set of functions. Thus, the structure has only one pointer to represent the behavior functions.

C++ compilers do this automatically. The virtual table is generated by the compiler for each type, and placed in constant memory. Here is an example, for the i960, of a call to a virtual method following a pointer in sfoo:

```
ld      _sfoo,g5
ld      8(g5),g4
ldis    8(g4),g0
ld      12(g4),g4
addo    g5,g0,g0
callx   (g4)
```

This generated code loads into g4 the pointer to the function to be execute, and into g0 the pointer to the object. The external pointer variable "`sfoo`" contains the pointer to the object to be manipulated, and the virtual method takes no parameters. With carefully crafted C code, the programmer can save maybe one instruction here.

Obviously, however, this code is far too much to use for acessor methods, such as "`Foo::value()`" above. Although the virtual method performs a useful function efficiently, it is clearly too expensive for trivial functions. The common technique of creating a virtual method that returns a constant value to reveal the true object type is inefficient. If you must do this, run time type information is more practical.

One unexpected benefit of virtual methods is that in certain cases the compiler can tell a priori which implementation of a virtual method applies, and can optimize all the above away, as in:

```
        ld      _sfoo,g0
        ld      _sfoo+4,g4
        cmpible g4,g0,L4
        mov     g4,g0
L4:
```

In this example, "sfoo" is an object and not a pointer. C++ knows exactly which implementation applies, and took the liberty of implementing the method inline. The only difference between this and the previous example is that "`sfoo`" is known exactly to be of type Foo.

4.3 Assembly Code

When dealing with "bare iron," some assembly code is inevitable. There is, for example, no reasonable way to implement thread switching entirely in C or C++, and in most processors interrupt and trap handlers must have assembly code to save the context of an interrupted thread and setup a fresh C/C++ calling sequence.

However, assembly code should typically be kept to a minimum. A human can do a good job of optimizing a small stretch of assembly code, but as the code grows, the human capacity to manage resource allocation becomes overwhelmed and the compiler performs better.

The paradox is that short assembly functions are more likely to be inefficient if placed in seperate source files, as the call and return overhead starts to overwhelm. What we really want is a way to write inline assembly code. Look at the following example:

```
inline int isr_hot_flag()
{
        register unsigned tmp;
        asm("modpc 0, 0, %0" : "=r" (tmp));
        return tmp & 0x2000;
}
```

This code returns true if the caller is running in an interrupt handler on an i960. The "`modpc`" instruction takes around 14 clock cycles to execute (it is slow) but that is not too bad. The `call` and `return` instructions on the i960Jx each consume 6 cycles and also push a register set, leading to at least 12 cycles for the call/return pair. Putting this function in a seperate source file would *double* the execution time of the function and may cause a register cache spill as well. Inlining this function is therefore rather important and powerful.

The benefits do not stop there. The GNU compiler syntax for inline assembly allows the programmer to match up registers and supply constraints that allow the C/C++ compiler to include the code in the optimization phase. The following degenerate case:

```
int foo() { return 0 && isr_hot_flag(); }
```

obviously leads to the following optimized i960 assembly code:

```
_foo__Fv:
        mov 0,g0
        ret
```

More complex situations are possible, but the point here is that the inline assembly can and should be included in the optimization passes of the compiler for the best results. A more complex example for the i386[7] works as follows:

```
inline int call_host1(int sys, int parm)
{
        asm ("int \$0x80\n"
                : "=a" (sys)
                : "0" (sys), "b" (parm) );
        return sys;
}
```

The previous code makes a system call by putting the parameter in the ebx register and the system call number in the eax register, and calling "`int $0x80`" to trap to the system. The compiler now knows how to allocate the eax and ebx registers around this assembly statement and can use that knowledge when optimizing register allocation around it. It can also eliminate the whole mess if the result is not at all used.[7]

This point doesn't have much to do with object oriented design (except that you can write methods in assembly code) but has everything to do with writing the low-level code in C++. If assembly code could not be inlined, an efficient implementation would necessarily pull more code into the assembly files to cut down on the cost of function call overhead. At that point, the C++ compiler would become more a hindrance then a help.

Including assembly code inline, and using constraints to control the optimizer, eliminates much of the interface overhead between C++ and assembly and gives the programmer the best of the C++ and assembly worlds.

4.4 Templates

This brings up an interesting theoretical optimization feature of templates. If in the previous example the class `Foo` were a template, the `size()` method could just return a template parameter. Thanks to template semantics, calls to the `size()` method are subject to constant elimination, which may in turn lead to dead code elimination. That is an example of the *compiler* deciding on the course of some branches, and eliminating the actual branch instruction from the execution stream.

Similar code in C uses preprocessor macros to get the same benefits, and is not type-safe. Templates used this way save much space and execution time, and are more readable. Use templates as a means of manipulating the type structure of the program, and not a way to create code, and templates may actually reduce to take no code space at all.

If template methods are not inlined, the compiler must generate an implementation of the template, often in a different compilation unit. To make matters worse, much near-duplicate code may be generated. For example, instantiations with the "`int`" type and the "`unsigned`" type may generate identical code for a small inlined method but generate unique implementations for complex out-lined methods. Comparisons, for example, require different assembly code for "`int`" and "`unsigned`" values.

Templates are therefore a terrific way to generate lots of excess code when used in this fashion. When inlined, the compiler may use template semantics to implement efficient, locally optimized code. Otherwise, they generate lots of redundant code.

Compiler writers are still arguing over how to deal with template repositories and the like, but in practice, for our purposes, the issue is moot. Large template classes or functions will expand to consume all available ROM. Practical templates should be small enough to be inline, and if inlined everywhere, there is no need for a template repository.

4.5 Smaller is Better

Software tends to expand to fill available memory, and programmers tend to judge their creations by the number of lines of code. There are two good reasons for worrying about program size, even when virtual memory is available and inexhaustible.

The most obvious reason to embedded programmers is that memory costs money and board space. Actually, it is the hardware designers who notice this first, and design in small amounts of memory. The programmer then asks for an additional 4Meg of flash memory and learns that 512Kbyte per chip means that 4Meg is 8 chips and the board just doesn't have that much space.

Large programs also tend to run slower. If it isn't simply because all those instructions take a long time to fetch from memory, it's because large programs overflow the instruction cache more often. Even dead code tends to spread the useful code into a larger address space and can lead to cache misses. Thanks to the widespread use of caches, memory

[7] If the assembly statement has side effects and should not be eliminated, it can be declared "`volatile`" and the compiler will preserve it.

and instruction, large programs are slow programs.

4.6 Link-time Efficiency

After all the compilation units are compiled, the program must be linked together with the run time library to create a single executable. On conventional operating systems, there are shared objects to be linked to, and the kernel to be made available. In a kernel-based operating system the common functions are placed in a protected kernel that all the applications share.

In an embedded environment, where there is only one application, the kernel is not being shared so its size should be included as part of the cost of the application. uCR is not kernelized, it is presented as a library that the linker uses to resolve symbols. Only the parts of uCR that are explicitly used are allocated space in the final image.

This is an example program, linked with uCR, that has only an empty main and the code necessary to enable all the devices on a Picture Elements ISE board. The text section includes executable code and constants and can be placed in ROM. The bss section is large because it contains generous stack space (10K) for the main thread.

```
text    data    bss     dec   filename
3520    1232    10664   15416 file.exe
```

The following is the same program compiled for Linux/SPARC. The stack space is not included in the totals, nor is the linux kernel itself or any of the 4 shared objects that this image links to.

```
text    data    bss     dec  filename
2688    2575    8       5271 a.out
```

Ignore the bss sections (mostly stack space in file.exe) and the linux image is about the same size. The linux image is for sparc whereas the uCR image is for i960 so the differences may easily be due to instruction set constraints.

What is significant about this example is that the entire uCR image takes up no more space then an image linked to run in an environment that has several megabytes of uncounted resources in the host kernel and shared objects. These resources provide a very important value in the case of Linux, but none of them are of interest in an embedded target.

As the program becomes more interesting, the uCR image sizes naturally increase. For example, the following numbers apply to a program that has included a debugger interface and code to drive a PCI bus interface, along with an implementation of a channel protocol that communicates with a driver on a host computer. In this image, 16K of buffer space is included in the bss number, along with the main stack.

```
text    data    bss     dec   filename
12544   2824    27284   42652 file.exe
```

The advantage of placing the uCR infrastructure in a library is that only the parts actually used are brought into the image. This can include individual methods of a class (those that are not inlined). A kernel image, on the other hand, must be included all at once or not at all.

5 Java, Anyone?

We have considered, and are still considering, the use of Java in embedded programming. There is an important problem with it, however, that reduces its usefulness, and that is the lack of an "asm" statement, and other inlining.

It turns out, when dealing with physical hardware, that there is always some little bit of assembly code that needs to be written. The obvious first thought is to put the assembly code is a library somewhere and call it when needed. But that is not necessarily the right answer. Consider the following familiar example for an Intel i960 microprocessor:

```
inline int  isr_hot_flag()
{
      register unsigned tmp;
      asm("modpc 0, 0, %0" : "=r" (tmp));
      return tmp & 0x2000;
}
```

This function basically reduces to the single instruction, the "modpc" instruction. Put that in a library and you get around it two branches and some register file shuffling, maybe even a few memory accesses. Wrap it up in a java class somewhere, and you also get the overhead of leaving and entering java.

A just-in-time compiler for java byte code would address many performance issues by generating unrolled directly executable machine code as the program runs. A syntax would be needed to specify specific assembly instructions for inclusion in the stream, or the compiler will not be able to match the optimization performance of C++.

6 Conclusions

uCR as a whole has become a richer environment, now that it works with more substantial processor boards. However, its original design goal to stay small and efficient seems to be working. The core library of uCR is still quite simple. We have also come to some conclusions on the matter of C++ and embedded programming.

We to this day face people telling us that C++ generates inefficient code that cannot possibly be practical for embedded systems where speed matters. The criticism that C++ leads to bad executable code is ridiculous, but at the same time accurate. Poor style or habits can in fact lead to awful results. On the other hand, a skilled C++ programmer can write programs that match or exceed the quality of equivilent C programs written by equally skilled C programmers.

The development cycle of embedded software does not easily lend itself to the trial-and-error style of programming and debugging, so a stubborn C++ compiler that catches as many errors as possible at compile time significantly reduces the dependence on run-time debugging, executable run-time support and compile/download/test cycles. This saves untold hours at the test bench, not to mention strain on PROM sockets.

7 Epilogue

There are times when the proper development environment for a project is not an operating system at all. For these times, run time suppport is all that is required for convenient development. If an operating system does not contribute to the solution, it is part of the problem and should not be there.

Without an operating system, however, the code generated must stand alone on the target hardware. The runtime support necessary to allow this, and even to add some extra features normally associated with operating systems (like threads and memory management) is fortunately not difficult or expensive.

uCR does a good job of providing the necessary runtime support to the compiler, with little overhead. Its small size comes from a careful attention to the details of performance, and stubborn controll of feature creep. The core design allows interesting functionality to be placed in libraries outside of uCR and brought in by the linker only if a programmer uses it.

With uCR, it is possible to get software for a new board up and running, if the CPU is already supported, in a few hours. It is an important milestone to get a trivial program running an infinite loop on a new processor board and it is best to not invest days getting there. Once trivial programs work, more extensive hardware debugging can commence.

The ongoing work on uCR is available for anonymous ftp from the Picture Elements ftp and web site, including the documentation, at:

`http://www.picturel.com/ucr/uCR.html`, and
`ftp://ftp.picturel.com/pub/source`.

It is indeed interesting that efforts to reduce code size and increase speed have led to the conclusion that significant portions of the source code must be visible to the programmer in the form of inline functions.

References

[1] American National Standaards Institute, 11 West 42nd Street, New York, New York 10036. *American National Standard for Programming Languages – C*, ansi/iso 9899-1990 edition, 1990.

[2] Grady Booch. *Object Oriented Design with Applications*, chapter 2, page 66. The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] Working paper for draft proposed international standard for information systems – programming language c++. http://www.cygnus.com/misc/wp/draft/index.html, April 1995.

[4] Cyclone Microsystems. *PCI Intelligent I/O Controllers User's Manual.*

[5] IEEE. *Information Technology–Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface.*

[6] Intel Corporation. *i960 RP Microprocessor User's Manual.*

[7] Intel Corporation. *Pentium Family User's Manual.*

[8] Intel Corporation. *MON960 Debug Monitor User's Guide*, 1995.

[9] Picture Elements, Inc. *Imaging Subsystem Engine - Theory of Operation.* Preliminary.

[10] P. J. Plauger. *The Draft Standard C++ Library.* Prentice Hall, 1995.

[11] PLX Technology. *PCI Bus Interface and Clock Distribution Chips.*

[12] Stephen Williams. Using ucr. http://www.picturel.com/ucr/uCR.html.

Implementing Optimized Distributed Data Sharing Using Scoped Behaviour and a Class Library

Paul Lu

Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, M5S 3G4
Canada
paullu@sys.utoronto.ca

Abstract

Sometimes, it is desirable to alter or optimize the behaviour of an object according to the needs of a specific portion of the source code (i.e., context), such as a particular loop or phase. One technique to support this form of optimization flexibility is a novel approach called *scoped behaviour*. Scoped behaviour allows the programmer to incrementally tune applications on a per-object and per-context basis within standard C++.

We explore the use of scoped behaviour in the implementation of the Aurora distributed shared data (DSD) system. In Aurora, the programmer uses scoped behaviour as the interface to various data sharing optimizations. We detail how a class library implements the basic data sharing functionality and how scoped behaviour coordinates the compile-time and run-time interaction between classes to implement the optimizations. We also explore how the library can be expanded with new classes and new optimization behaviours.

The good performance of Aurora suggests that using scoped behaviour and a class library is a viable approach for supporting this form of optimization flexibility.

1 Introduction

Optimizing a program's data access behaviour can significantly improve performance. Ideally, the programming system should allow each object to be optimized independently of other objects and each portion of the source code (i.e., context) to be optimized independently of other contexts. Towards that end, researchers have explored various compiler and run-time techniques to provide per-object and per-context flexibility in applying an optimization.

We describe how *scoped behaviour*, a change in the implementation of methods for the lifetime of a language scope, can provide the desired optimization flexibility within standard C++. A language scope (i.e., nested braces in C++) around source code selects the context and the re-defined methods implement the optimization. Scoped behaviour requires less engineering effort to implement than compiler extensions and it is better integrated with the language, thus less error-prone to use, than typical run-time libraries.

Specifically, we focus on a single application of scoped behaviour: supporting optimized distributed data sharing. Since this discussion is closely tied to a particular problem domain, we begin with a brief introduction to distributed data sharing. Then we provide an overview of the Aurora distributed shared data system [Lu97], detail how scoped behaviour and the class library are implemented, and discuss some performance issues.

2 Distributed Data Sharing

Parallel programming systems based on shared memory and shared data models are becoming increasingly popular and widespread. Accessing local and remote data using the same programming interface (e.g., reads and writes) is often more convenient than mixing local accesses with explicit message passing.

On distributed-memory platforms, the lack of hardware support to directly access remote memories has prompted a variety of software-based, logically-shared systems. Broadly speaking, there are *distributed shared memory* (DSM) [Li88, BCZ90, ACD+96] and *distributed shared data* (DSD) [BKT92, SGZ93, JKW95] systems. Support for distributed data sharing, whether it is page-based as with DSM, or object-based (or region-based) as with DSD, is an active area of research. The spectrum of implementation techniques spans special hardware support, run-time function libraries, and special compilers.

Layer	Main Components and Functionality
Programmer's Interface	Teams of threads for SPMD-style parallelism, active objects Distributed vector and scalar objects *Scoped behaviour*
Shared-Data Class Library	Handle-body shared-data objects Overloaded operators and special methods; immediate data access (default behaviour) Data sharing optimizations Owner-computes, caching data for reads, release consistency for writes
Run-Time System	Active objects and remote method invocation (currently, ABC++) Threads (currently, pthreads) Communication mechanisms (currently, shared memory and MPI)

Table 1. Layered View of Aurora

(a) Original Loop

```
GVector<int> vector1( 1024 );



for( int i = 0; i < 1024; i++ )
  vector1[ i ] = someFunc( i );
```

(b) Optimized Loop Using Scoped Behaviour

```
GVector<int> vector1( 1024 );

{ // Begin new language scope
  NewBehaviour( vector1, GVReleaseC, int );

  for( int i = 0; i < 1024; i++ )
    vector1[ i ] = someFunc( i );
} // End scope
```

Figure 1. Applying a Data Sharing Optimization Using Scoped Behaviour

In this context, the all-software Aurora DSD system provides a shared-data programming model on distributed-memory hardware. All shared data are encapsulated as objects and are accessed using methods. To overcome the latency and bandwidth performance problems of typical distributed-memory platforms, Aurora provides a set of well-known data sharing optimizations.

Although other DSM and DSD systems also offer data sharing optimizations, Aurora is unique in how these optimizations are integrated into the programming language. Pragmatically, scoped behaviour allows the applications to be incrementally tuned with reduced programmer effort. Also, as an experimental platform, Aurora's class library approach is relatively easy to extend with new behaviours. In particular, one of the goals of this research is to support common data sharing idioms, specified and optimized using scoped behaviour, with good performance.

3 Overview of Aurora

Aurora can be viewed as a layered system (Table 1). The key layers will be discussed later on, but we begin with a quick overview.

Application programmers are primarily concerned with the upper two layers of the system: the programmer's interface and the shared-data class library. The basic data-parallel process model is that of teams of threads operating on shared data in SPMD-fashion (single program, multiple data). The basic shared-data model is that of a distributed vector object or a distributed scalar object. Once created, a shared-data object is transparently accessed, regardless of the physical location of the data, using normal C++ syntax. By default, shared data is read from and written to immediately (i.e., synchronously), even if the data is on a remote node, since that data access behaviour has the least error-prone semantics.

Figure 1(a) demonstrates how a distributed vector object is instantiated and accessed. `GVector` is a C++ class template provided by Aurora. Any built-in data type or user-defined structure or class can be used as the template argument. The size of the vector is a parameter to the constructor and, currently, the vector elements are block distributed across the physical nodes.

Now, for example, if a shared vector is updated in a loop *and* if the updates do not need to be performed immediately, then the loop can use release consistency [GLL+90, AG96] and batch the writes (see Figure 1(b), shown side-by-side for easy comparison). Without any changes to the loop code itself, the *behaviour* of the updates to `vector1` is changed within the language scope.

(a) Common Preamble

```
int i, j;
// Prototype of C-style function with innermost loop
int dotProd( int * a, int * b, int j, int n );
```

(b) Sequential Code

```
// mA, mB, mC are 512 × 512 matrices

for( i = 0; i < 512; i++ )
  for( j = 0; j < 512; j++ )
    mC[i][j] =
      dotProd( &mA[i][0], mB, j, 512 );
```

(c) Optimized Parallel Code

```
// mA, mB, mC are 512 × 512 GVectors

{ // Begin new language scope
 NewBehaviour( mA, GVOwnerComputes, int );
 NewBehaviour( mB, GVReadCache, int );
 NewBehaviour( mC, GVReleaseC, int );

 while( mA.doParallel( myTeam ) )
   for( i = mA.begin();i < mA.end();i += mA.step())
     for( j = 0; j < 512; j++ )
       mC[i][j] =
         dotProd( &mA[i][0], mB, j, 512 );
} // End scope
```

Figure 2. Matrix Multiplication in Aurora

The `NewBehaviour` macro specifies that the release consistency optimization should be *applied* to `vector1`.

Therefore, scoped behaviour is the main interface between the programming model and the data sharing optimizations, providing:

- *Per-object* flexibility: The ability to apply an optimization to a specific shared-data object without affecting the behaviour of other objects. Within a context, different objects can be optimized in different ways (i.e., heterogeneous optimizations).
- *Per-context* flexibility: The ability to apply an optimization to a specific portion of the source code. Different portions of the source code (e.g., different loops and phases) can be optimized in different ways.

The lowest layer of Aurora, the run-time system, provides the basic thread management and communication mechanisms. The current implementation of Aurora uses the ABC++ class library for its *active object* mechanism, an object that has a thread of control associated with it, and *remote method invocation* (RMI) facilities [OEPW96]. RMIs are syntactically similar to normal method invocations, but RMIs can be between objects in different address spaces. If desired, the application programmer can directly utilize the active object and RMI mechanisms to implement a more control-parallel process model. Also, although ABC++ already has a parametric shared region (PSR) mechanism, it is not used by Aurora.

In turn, ABC++ uses standard pthreads [Pth94] for concurrency and either shared memory or MPI message passing [GLS94] for communication.

4 Programmer's Interface

A more detailed description of the programmer's interface to Aurora can be found elsewhere [Lu97], but we briefly touch upon the main ideas with an example.

4.1 Example: Matrix Multiplication

For illustrative purposes, consider the problem of non-blocked, dense matrix multiplication, as shown in Figure 2. The preamble is common to both the sequential and parallel codes (Figure 2(a)). The basic algorithm consists of three nested loops, where the innermost loop computes a dot product and can be factored into a separate C-style function. An appropriate indexing function for two-dimensional arrays in C/C++ is assumed.

Conceptually, we can view an optimization as a change in the type of the shared object for the lifetime of the scope. The current set of available behaviours is summarized in Table 2. As an example of per-object flexibility, three different data sharing optimizations are applied to

Scoped Behaviour	Description
Owner-computes	Threads access only co-located data.
Caching for reads	Create local copy of data.
Release consistency	Buffer write accesses.
Special-purpose data movement	Used with owner-computes for specific applications (e.g., stencils in 2-D diffusion simulation).

Table 2. Some Scoped Behaviours

the sequential code in Figure 2(b) to create the parallel code in Figure 2(c). Specifically:

1. `NewBehaviour(mA,GVOwnerComputes,int)`: To partition the parallel work, the owner-computes technique is applied to distributed vector `mA`.

   Within the scope, `mA` is an object of type `GVOwnerComputes` and has special methods `doParallel()`, `begin()`, `end()`, and `step()`. Only the threads (each represented by a local `myTeam` pointer) that are co-located with a portion of `mA`'s distributed data actually enter the `while`-loop and iterate over their local data. Also, when `dotProd()` is called, a type constructor for `GVOwnerComputes` returns a C-style pointer to the local data so that the function executes with maximum performance.

   Although some changes to the source code are required to apply owner-computes, they are relatively straightforward. Other work partitioning strategies, that do not use the special methods provided by Aurora, are allowed, but owner-computes is both convenient and efficient.

2. `NewBehaviour(mB, GVReadCache, int)`: To automatically create a local copy of distributed vector `mB` at the start of the scope, since it is read-only and re-used many times, its type is changed to `GVReadCache`.

   The scoped behaviour of a read cache also includes a type constructor so that `dotProd()` can be called with C-style pointers that point to the cache. Note that no lexical changes to the loop's source code are required for this optimization.

3. `NewBehaviour(mC, GVReleaseC, int)`: To reduce the number of update messages to elements of distributed vector `mC` during the computation, its type is changed to `GVReleaseC`.

   Within the scope, the overloaded operators batch the updates into a per-target address space buffer and messages are only sent when the buffer is full or when the scope is exited. Also, multiple writers to the same distributed vector are allowed. No lexical changes to the source code are required.

The result of this heterogeneous set of optimizations is that the nested loops can execute without remote data accesses and the parallel program can use the same efficient `dotProd()` function as in the sequential program.

4.2 Discussion: Programming in Aurora

The typical methodology for developing Aurora applications consists of three main steps. First, the code is ported to Aurora. Shared arrays and shared scalars are converted to `GVectors` and `GScalars`. Although the default immediate access policy can be slow, its performance can be optimized after the program has been fully debugged.

Second, the work is partitioned among the processors and threads. Owner-computes and SPMD-style parallelism are common and effective strategies for many applications. However, the application programmer may implement other work partitioning schemes.

Lastly, various data sharing optimizations can be tried on different bottlenecks in the program and on different shared-data objects. Often, the only required changes are a new language scope and a `NewBehaviour` macro. Sometimes, straightforward changes to the looping parameters are needed for owners-computes. For example, in the matrix multiplication program, owner-computes can be applied to vector `mC` instead, with read caches used for both vector `mA` and vector `mB`. The `dotProd()` function and the data access source code remain unchanged. The new optimization strategy uses more resources for read caches than the original strategy, but, since `mC` is being updated, it is perhaps a more conventional application of owner-computes. Reverting back to the original strategy is also relatively easy. For the application programmer, the ability to experiment with different optimizations, with limited error-prone code changes, can be valuable.

5 Scoped Behaviour

Scoped behaviour is a change in the implementation of selected methods for the lifetime of a language scope.

For the Aurora programmer, scoped behaviour is how an optimization is applied to a shared-data object. For the system and class designer, scoped behaviour is an interface between collaborating classes that changes the implementation of the selected methods. Some of the ideas behind scoped behaviour have been explored as part of the handle-body and envelope-letter idioms in object-oriented programming [Cop92] (to be discussed

(a) Scoped Behaviour Macro

```
#define NewBehaviour( XX, YY, ZZ ) \                    // Macro provided by aurora.H
  GPortal<GVector<ZZ> > AU_ ## XX( XX ); \
  YY<ZZ> XX( AU_ ## XX );

template <class C_OrigHandle>                           // Class template provided by aurora.H
class GPortal
{
  private:
    C_OrigHandle * save;                                // Saved handle
  public:
    GPortal( C_OrigHandle & h ) { save = &h; }          // In: Constructor
    operator C_OrigHandle &() { return *save; }         // Out: Type constructor
}; // GPortal
```

(b) Source Code

```
{ // Begin new language scope
  NewBehaviour( vector1, GVReleaseC, int );


  for( int i = 0; i < 1024; i++ )
    vector1[ i ] = someFunc( i );
} // End scope
vector1[ 0 ] = 1;                    // Immediate update
```

(c) After Standard Preprocessor Pass

```
{ // Begin new language scope
  GPortal<GVector<int> > AU_vector1( vector1 );
  GVReleaseC<int> vector1( AU_vector1 );

  for( int i = 0; i < 1024; i++ )
    vector1[ i ] = someFunc( i );
} // End scope
vector1[ 0 ] = 1;                    // Immediate update (still)
```

Figure 3. Aurora's Scoped Behaviour Macro

further in Section 6.1). Scoped behaviour builds upon these ideas.

5.1 Language Scopes and Scoped Behaviour Objects

The main motivation for using language scopes to define the context of scoped behaviour is to exploit the property of *name hiding*. In block-structured languages, an identifier can be re-used within a nested language scope, thus hiding the identifier outside of the scope.

Instantiations of a class that are designed to be used within a language scope, and which hide objects outside the scope, are called *scoped behaviour objects*.

5.2 Implementing Scoped Behaviour

As shown in Figure 3(a), Aurora provides the scoped behaviour macro `NewBehaviour` and the class template `GPortal` via a header file. Figure 3(b) shows the original programmer's source code and Figure 3(c) shows the code after the standard preprocessor of the C++ compiler has expanded the macro. Again, the code is shown side-by-side for comparison.

The `NewBehaviour` macro is parameterized by the name of the original shared-data object, the type of the new scoped behaviour object, and the type of the vector elements.[1] The macro instantiates two objects. The first object, `AU_vector1`, is of type `GPortal`. Its sole function is to cache a pointer to the original object, which is passed as a constructor argument, and then pass it along to the scoped behaviour object's constructor. The second object, the scoped behaviour object `vector1` of type `GVReleaseC<int>`, hides the original object but can access its internal state using the pointer passed by `AU_vector1`. Thus, the scoped behaviour object can mimic or change the functionality of the original shared-data object.

We will discuss the implementation of these classes in more detail in Section 6, but we provide an overview of the basic ideas.

Since the scoped behaviour object has the same name as the original `vector1`, the compiler will generate

[1] Note that it is a multi-line macro and the `##` symbol is the standard preprocessor operator for lexical concatenation. Also, the prefix `AU_` is arbitrary and can be redefined, if necessary.

Unfortunately, the more concise syntax of `GVReleaseC<int> vector1( vector1 )` conflicts with the C++ standard (i.e., the new `vector1` is passed a reference to itself, instead of to the original object), so an intermediary object is required. Fortunately, the macro hides the existence of the intermediary object.

the loop body code according to class `GVReleaseC` instead of the original object's class. However, the user's source code does not change. Even though the original and scoped behaviour objects collaborate to implement scoped behaviour, we can conceptualize it as temporarily changing the type of the original object. The `NewBehaviour` macro helps to hide this abstraction. Note that source code outside of the context of the optimization continues to refer to the original `GVector`. Therefore, immediate update remains the default behaviour outside of the scope, illustrating per-context flexibility.

The class template `GVReleaseC` is designed to behave exactly like `GVector`, except that the overloaded operators now buffer updates to the vector elements. Read accesses to the vector continue to be performed immediately, even if the data is remote. Thus, the class of a scoped behaviour object can selectively redefine behaviour on a method-by-method and operator-by-operator basis.

Also, since `vector1` is a new object within the scope, dynamic run-time actions can be associated with the various constructors and the destructor. In particular, the destructor flushes the update buffers to the vector so that all updates are guaranteed to be performed when the scope is exited.

Although this description has centered on a particular class, the basic scoped behaviour technique can be applied to a variety of classes and objects. The owner-computes, caching for reads, and other behaviours use the same `NewBehaviour` macro and are based on the same design principles.

Of course, the basic ideas behind the implementation of scoped behaviour are not new. The notion of nested scopes is fundamental to block-structured sequential languages. The association of data movement actions with C++ constructors and destructors is also not new (for example, in ABC++). However, scoped behaviour is unique in that it coordinates the interaction of different classes to create per-object and per-context behaviours.

5.3 Advantages and Disadvantages

The advantages of scoped behaviour include:

1. *Standards-based implementation.* Scoped behaviour can be implemented within standard C++ as a preprocessor macro. The class library, to be discussed in the next section, is also standard C++.

2. *Flexibility of experimentation.* Scoped behaviour makes it easy to add, modify, and remove behaviours with minimal or no lexical source code changes.

3. *Flexibility of implementation.* The compile-time aspect of scoped behaviour allows the compiler (and implementor) to generate behaviour-specific code based on different classes. The run-time aspect of scoped behaviour allows dynamic behaviour, such as data movement and interactions with the run-time system, to be associated with constructors and destructors.

A disadvantage of scoped behaviour is that, since it is a programming technique instead of a first-class compiler feature, it cannot access the compiler's symbol table for high-level analyses. A more general disadvantage is that, since the run-time behaviour depends on constructors and destructors with static invocation points, it cannot be directly ported to a language like Java [Sun96]. Java is a garbage-collected language and the current definition does not have destructors in the same sense as C++.

Compared to some other DSM and DSD systems, scoped behaviour has safety and performance benefits.

For example, `GVReleaseC` has been explicitly implemented with a constructor that takes a parameter of type `GVector&`. Therefore, programming errors involving incompatible objects, such as trying to use release consistency with normal C++ arrays, will result in compile-time errors. More generally, as with all object-oriented systems, methods are invoked on objects and thus it is impossible to pass the wrong shared-data object as a function call parameter. Also, the automatic construction and destruction of scoped behaviour objects make it impossible for the programmer to omit a required data movement action at the end of a context. Non-object-oriented function libraries may only be able to catch these forms of errors at run-time, if at all.

As with some other systems, performance benefits can arise from exploiting high-level data access semantics. For example, `GVReadCache` is intended for data that is read-only and where most of the elements will be accessed during the context. Therefore, Aurora can read the data in bulk rather than demanding-in each portion of the data with a separate data movement action. Also, `GVReleaseC` is intended for data that is updated but not read. Therefore, unlike some other systems, Aurora can avoid the overhead of demanding-in the remote data before overwriting it.

6 Shared-Data Class Library

In this section, we take a detailed look at the design and implementation of the C++ classes for the shared-data objects and data sharing optimizations. By design, these classes collaborate to support scoped behaviour.

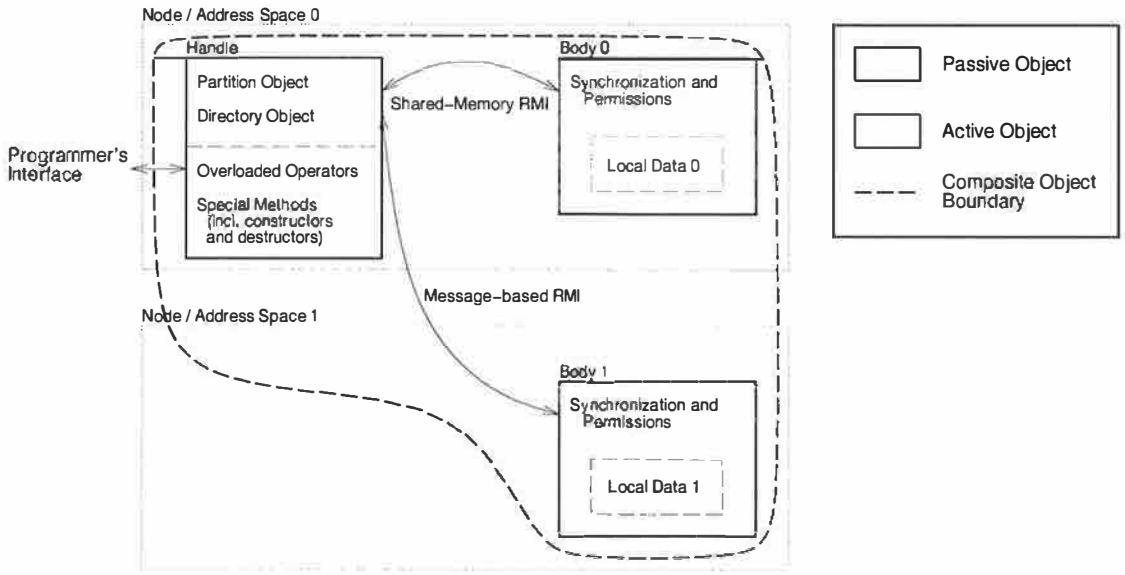


Figure 4. Handle-Body Composite Objects

6.1 Handle-Body Composite Objects

The main architectural feature of the shared-data class library is the use of the handle-body idiom to create *composite objects* [Cop92, OEPW96] for shared data (Figure 4). The *handle object* defines the programmer's interface to the shared data. The *body object* (or objects) contain the actual data.

The extra level of indirection afforded by a composite handle-body approach allows for:

1. *Data distribution.* A distributed vector is a set of body objects and each body object can be located in a different address space or on a different physical node. The handle includes a *partition object* to abstract the distribution strategy and a *directory object* to keep track of the location of the bodies. A distributed scalar has a single body object.

   Figure 4 shows a distributed vector object with a handle and two body objects, where one of the body objects is on a different node than the handle.

2. *Location-transparent data accesses.* Through overloaded operators in the handle, the distributed data can be accessed through a uniform interface, regardless of the location of the actual data. Thus, for a given vector index, the partition object determines which body holds the data and the directory object provides a pointer to the body object.

3. *Cheap parameter passing of shared data.* Only handles are passed across function calls; the data in the bodies are not copied. Handles can also be passed between address spaces, if desired, since the partition and directory objects are sufficient to locate any body object from any address space.

For performance-sensitive functions, such as `dotProd()` in Figure 2, the overheads of indirection can be avoided in controlled ways through type constructors that return C-style pointers.

The current implementation of Aurora creates handles as passive (i.e., regular) C++ objects. However, each individual body is implemented as an active object, which is useful for implementing any necessary synchronization behaviour. Handle and body interact using remote method invocations. The run-time system automatically selects between shared-memory and message-based communication mechanisms for transmitting RMIs.

6.2 Class Hierarchy for Handles

Since most of the data sharing functionality is implemented in the handles, this discussion will focus on the handle classes. Briefly, however, the body classes support `get()` and `put()` data access methods, including batch update and block-read variations. For the current data sharing optimizations in Aurora, this simple functionality is all that is required.

Figure 5 is a diagram of the main classes in the class hi-

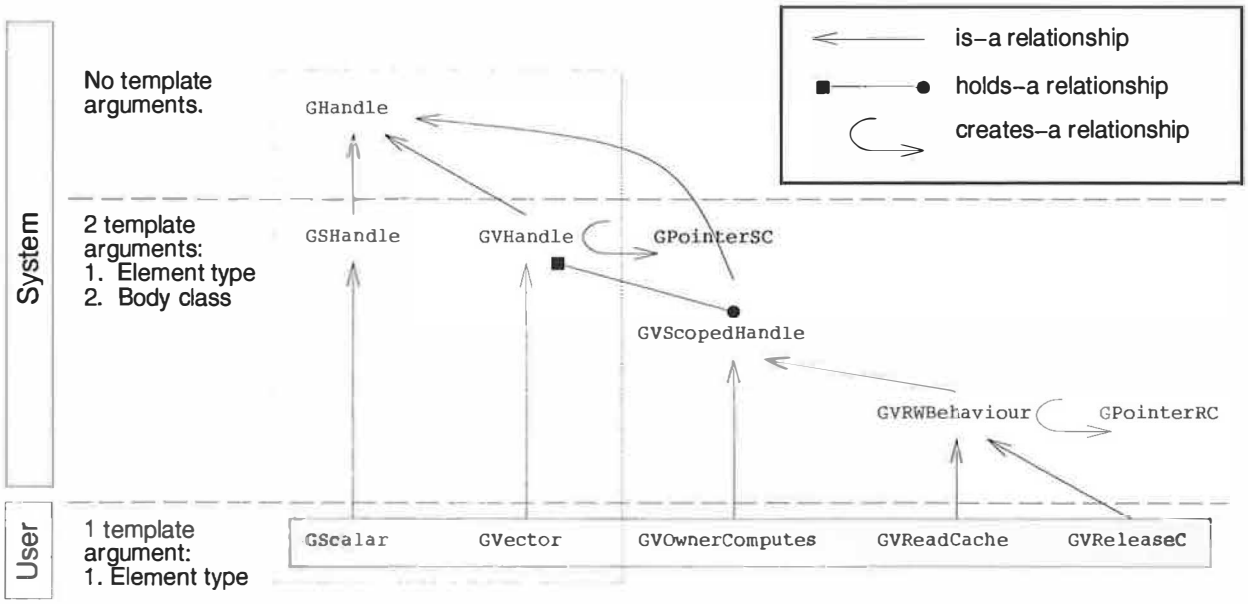


Figure 5. Class Hierarchy for Handles

erarchy for shared-data handles.[2] Aside from the names of the classes, the diagram shows the relationship between classes. The *is-a* relationship is the usual notion of inheritance. For example, class `GHandle` is the base class for all handles. Common access methods are factored into the base class. The *holds-a* relationship exists when a class contains a pointer (or pointers) to an instance of another class. This is used, for example, to allow one object to access the internal state of another object. The *creates-a* relationship exists when at least one of the methods of a class returns an object of another class. For example, an overloaded subscript operator (i.e., `operator[]`) can return an object which encodes information about a specific vector element [Cop92].

We can also distinguish the classes by the way they are, or are not, templated. Class `GHandle` is not templated in order to simplify the implementation of mechanisms that only require limited functionality from a handle. For example, querying about the number of vector elements does not require knowledge about template arguments. However, the most important class templates for the system implementor are parameterized by both the data element type and the class of the body object.

In general, the application programmer is only expected to use the classes with a single template argument for the data element type (labelled "User" in Figure 5 and highlighted in gray). These classes hide the more complex templating and class hierarchy considerations that the "System" must deal with.

For data sharing using immediate access, the important classes are `GSHandle` and `GVHandle` (shown inside the box in Figure 5). These classes encapsulate member data to keep track of the body or bodies.

Figure 6 provides a more detailed look at the interfaces for the classes that implement the shared vector. Class `GHandle`, which is not templated, is a convenient base class within which to implement methods common to all handles. Class `GVector` does little more than specify the specific body class (i.e., `LVector`) for the second template argument to `GVHandle` and call the appropriate constructors.

Most of the functionality for the shared vector is implemented by class `GVHandle`. In particular, the overloaded subscript operator returns an object of type `GPointerSC`, which is a *pointer object*. When evaluating C++ expressions involving objects and overloaded operators, temporary objects represent the result of sub-expressions. Since the actual data for a term may be a remote shared data element, the temporary object points to the body object with the data. Class `GPointerSC` has data members to store the vector index and a pointer to the specific body object with that element. Reading from or writing to the vector element invokes the appropriate type constructors and the overloaded assignment operator of `GPointerSC`, resulting in an immediate remote memory access.

[2] The notation is based on Booch [Boo91], but with some simplifications and changes to better suit this presentation.

```
// Base class. Not templated.
class GHandle
{
  private:
    int numElements;                                        // Number of vector elements
    // ...other data members...
  public:
    // ...various constructors and destructor...
    int size() { return numElements; }                      // Common access method
    // ...other methods...
}; // GHandle (System)


// Template argument C_Data is the element type; C_LV is the body class.
//  Classes GVScopedHandle, Partition, Directory, GPointerSC are provided by Aurora.
template <class C_Data, class C_LV>
class GVHandle : public GHandle                             // is-a GHandle
{
    // GVScopedHandle needs access to internal state (for holds-a)
    friend GVScopedHandle<C_Data, C_LV>;
  protected:
    Partition<MAX_LOCALS> partition;                        // Distribution strategy
    Directory<C_LV> directory;                              // Location of body object(s)
    // ...other data members...
  public:
    GVHandle( int numElements );                            // Construct with size of vector
    ~GVHandle();
    GPointerSC<C_LV, C_Data> operator[] ( int index );      // Immediate data access (creates-a)
    // ...other methods...
}; // GVHandle (System)


// Template argument C_Data is the element type; LVector (provided by Aurora) is the body class.
template <class C_Data>
class GVector : public GVHandle<C_Data, LVector<C_Data> >   // is-a GVHandle
{
  public:
    GVector( int numElements ) :                            // Construct with size of vector
          GVHandle<C_Data, LVector<C_Data> >( numElements ) {}
    ~GVector();
    // ...inherits operator[] and other methods...
}; // GVector (User)
```

Figure 6. Interface for Shared Vector: `GVector`

```
// Template argument C_Data is the element type; C_LV is the body class.
//  Remember that I am a friend of GVHandle.
template <class C_Data, class C_LV>
class GVScopedHandle : public GHandle                                   // is-a GHandle
{
  protected:
    GVHandle<C_Data, C_LV> * origHandle;          // To access internal state of original object (holds-a)
    // ...other data members...
  public:
    GVScopedHandle( GVHandle<C_Data, C_LV> & gv )                       // Construct with original handle
          [ origHandle = &gv; ]                                         // Cache the handle
    ~GVScopedHandle();
    // ...other methods...
}; // GVScopedHandle (System)


// Template argument C_Data is the element type; C_LV is the body class.
//  Classes Cache, BatchWrite, and GPointerRC are provided by Aurora.
template <class C_Data, class C_LV>
class GVRWBehaviour : public GVScopedHandle                             // is-a GVScopedHandle
{
  protected:
    Cache<C_Data, C_LV> * readCache;                                    // Configurable read cache
    BatchWrite<C_Data, C_LV> * updateBuf[MAX_LOCALS];   // Configurable buffer for release consistency
    // ...other data members...
  public:
    GVRWBehaviour( GVHandle<C_Data, C_LV> & gv ) :                      // Construct with original handle
          GVScopedHandle<C_Data, C_LV> >( gv ) {}
    ~GVRWBehaviour();                                   // Destructor flushes update buffers if necessary
    createCache();                                                      // Method to create read cache
    allowUpdateBuf();                                                   // Method to allow update buffers
    GPointerRC<C_LV, C_Data> operator[] ( int index );     // Data access via cache/buffer (creates-a)
    // ...other methods...
}; // GVRWBehaviour (System)


// Template argument C_Data is the element type; LVector (provided by Aurora) is the body class.
template <class C_Data>
class GVReleaseC : public GVRWBehaviour<C_Data, LVector<C_Data> >       // is-a GVRWBehaviour
{
  public:
    GVReleaseC( GVector<C_Data, C_LV> & gv ) :     // Original handle via GPortal of NewBehavourmacro
          GVRWBehaviour<C_Data, LVector<C_Data> >( gv )
          { allowUpdateBuf(); }                                         // Construct to allow update buffers
    ~GVReleaseC();
    // ...inherits operator[] and other methods...
}; // GVReleaseC (User)
```

Figure 7. Interface for Release Consistency Scoped Behaviour: `GVReleaseC`

6.3 Data Sharing Optimizations: Scoped Behaviour Objects

For the data sharing optimizations, the parent class `GVScopedHandle` extracts and maintains information about the internal state of a given `GVHandle`, as per the holds-a relationship (Figure 7). This functionality is an important part of implementing scoped behaviour. The partition and directory objects of the `GVHandle` are not copied, thus reducing the construction costs of a scoped behaviour object.

Class `GVOwnerComputes`, in its constructor, uses the extracted internal state to determine the address of the body object's data. Therefore, `GVOwnerComputes` can return a C-style pointer from the appropriate type constructor and from the overloaded subscript operator. As previously discussed, `GVOwnerComputes` also defines special functions to support easy iterating over the local data.

Class `GVRWBehaviour` can, optionally, create a read cache for shared data and create update buffers to shared data (Figure 7). Classes that derive from `GVRWBehaviour` explicitly configure the caching and buffering options. The overloaded subscript operator in `GVRWBehaviour` returns an object of class `GPointerRC`, which is similar in concept to class `GPointerSC`, but with two important differences. First, if the read cache exists and is loaded, then `GPointerRC` is configured to access data from the cache instead of from the remote body. Second, if the update buffers are enabled in `GVRWBehaviour`, then `GPointerRC` is configured to store updates in the buffer rather than initiate a remote memory access. `GVRWBehaviour` creates the buffers on demand. Depending on the configuration of the cache and buffers, `GPointerRC` will access shared data appropriately.

Therefore, the constructor of class `GVReadCache` calls the appropriate `GVRWBehaviour` methods to create and load the read cache. Thus, when the subscript operator for `GVReadCache`, which is inherited from the parent class, creates a `GPointerRC` object, it will always access the cache. `GVReadCache` also defines a type constructor to return a C-style pointer to the cache.

Similarly, class `GVReleaseC` calls the appropriate `GVRWBehaviour` constructor and enables the use of update buffers (Figure 7). Thus, when the subscript operator for `GVReleaseC`, which is inherited from the parent class, creates a `GPointerRC` object, it will always use the buffers. The destructor for class `GVRWBehaviour` makes sure all buffers are flushed.

7 Extending the Library

Within the class hierarchy, new data sharing optimizations can be implemented. We consider a trivial but illustrative example. For example, a new class could both cache data for reading *and* buffer updates. The new class would derive from `GVRWBehaviour`. The new class's constructor creates the read cache and also enables the update buffers. The `GPointerRC` objects created by the new class would always read from the cache and always buffer updates. By default, updates are also mirrored in the cache. Admittedly, this "new" data sharing optimization is easy to add because of the design and existing functionality of `GVRWBehaviour` and `GPointerRC`, but the basic techniques can be used for more complex additions to the library.

There are three main techniques for extending the library of data sharing optimizations. The techniques can also be combined.

1. *New classes.* Define new classes for partition, directory, body, and pointer objects.

   Currently, only a block-distributed partition object is implemented. If a cycle-distributed object is required in the future, a new partition class could abstract the distribution details. Finally, as we have seen, classes like `GPointerSC` and `GPointerRC` are useful for defining new memory access behaviours.

2. *New methods.* Inherit from a parent class, then add new scoped behaviour with new methods.

   For example, `GVOwnerComputes` adds new methods for iterating over local data.

3. *Re-define methods.* Inherit from a parent class, then re-define behaviour through constructors, the destructor, methods, operators, and type constructors.

   For example, `GVReleaseC` relies on its parent class for most of its functionality. `GVReleaseC` merely configures the update buffers appropriately in its constructor.

8 Performance

To date, we have experimented with three Aurora programs [Lu97]. The programs are matrix multiplication (Figure 2), a 2-D diffusion simulation, and Parallel Sorting by Regular Sampling (PSRS) [SS92, LLS+93]. Recent performance results are shown in Table 3. Speedups are computed against C implementations of the same algorithm (or against quicksort in the case of the parallel

Program	Data Set	Network	Speedup		
			2 PEs	4 PEs	8 PEs
Matrix Multiply	704 × 704 (175 sec. seq.)	Fast Ethernet	1.85	3.51	6.40
	512 × 512 (65.8 sec. seq.)	Fast Ethernet	1.79	3.37	5.89
2-D Diffusion	1526 × 1526, 32 time-steps (47.8 sec. seq.)	Fast Ethernet	1.27	2.13	3.86
	1024 × 1024, 32 time-steps (20.3 sec. seq.)	Fast Ethernet	1.07	1.91	3.45
PSRS	10 million keys (60.4 sec. seq.)	Fast Ethernet	n/a	2.24	3.72
	6 million keys (33.9 sec. seq.)	Fast Ethernet	1.21	2.05	3.22

Table 3. Aurora Programs on a Network of Workstations

sort). In particular, the sequential implementations do not suffer from the overheads of either operator overloading or scoped behaviour.

The distributed-memory platform used for these experiments is a cluster of PowerPC 604 workstations with 133 MHz CPUs, 96 MB of main memory, and a single, non-switched 100 Mbit/s Fast Ethernet network. The software includes IBM's AIX 4.1 operating system, AIX's pthreads, and the MPICH (version 1.0.13) [DGLS93] implementation of MPI.

Two trends can be noted in the performance results. First, for these three programs, additional processors improves speedup, albeit with diminishing returns. Second, as the size of the data set increases, the overall granularity of work, and thus speedup, also increases.

Contention for the single network and a reduced granularity of work can account for the diminishing returns for more processors with a fixed problem size. For example, since the read cache's data requirements are constant per-processor, communication costs and network contention grows when replicating vector `mB` in matrix multiplication. Communications costs under contention also account for the overheads in the parallel sort program, since the algorithm includes a key exchange. For the 2-D diffusion simulation, the granularity of a time-step before a barrier quickly falls to below one second as processors are added. Fortunately, if the problem size increases, the computation's overall granularity also increases resulting in better absolute speedups.

The performance of Aurora programs on this particular hardware platform is encouraging, but there remains two important avenues for future work: different network technology and new scoped behaviours. An 155 Mbit/s ATM network has been installed on the platform, but it is not yet fully exploited by the run-time system. However, early experience indicates that the additional bandwidth and improved contention characteristics of ATM will benefit Aurora programs. Also, there is currently no overlap between communication (for reads) and computation in the existing scoped behaviours. For simplicity, `GVReadCache` loads all of the data before allowing computation to continue. Using the techniques described in this paper, the library of scoped behaviours will be extended to better hide the read latency of the distributed-memory hardware.

9 Discussion and Related Work

Distributed data sharing is an example of a problem domain where per-object and per-context optimization flexibility is desirable. The data access behaviour of a shared-data object can change depending on the loop or program phase, so a single data sharing policy is often insufficient for all contexts. In general, optimization flexibility can be supported through compiler annotations or a run-time system interface, but scoped behaviour offers advantages in terms of engineering effort, safety, and implementation flexibility.

Since Ivy [Li88], the first DSM system, a large body of work has emerged in the area of DSM and DSD systems (for example, [BCZ90, BKT92, BZS93, SGZ93, JKW95, ACD+96]). Related work in parallel array classes (for example, [LQ92]) has also addressed the basic problem of transparently sharing data.

Different access patterns on shared data can be optimized through type-specific protocols and run-time annotations. Both Munin [BCZ90] and Blizzard [FLR+94] provide protocols customized to specific data sharing behaviours. Run-time libraries, such as shared regions [SGZ93], SAM [SL94], and CRL [JKW95], associate

coherence actions with access annotations (i.e., function calls). Unlike Munin, Aurora does not require special compiler support and different optimizations can be used in different contexts. Unlike Blizzard, Aurora integrates the optimizations into the programming language to generate custom code for different coherence actions, for added implementation and performance flexibility. Unlike function libraries, the automatic construction and destruction of scoped behaviour objects make it impossible for the programmer to omit an annotation and miss a coherence action.

Aurora's handle-body object architecture and the association of data movement with constructors and destructors are inspired by the parametric shared region (PSR) mechanism of ABC++. However, there are some significant differences between Aurora's shared-data objects and PSRs. First, Aurora allows distributed vectors to be partitioned between different address spaces to improve scalability and to support owner-computes using multiple nodes. A PSR has single home node, therefore shared data cannot be partitioned and owner-computes cannot be used within a PSR. Second, Aurora uses operator overloading and pointer objects, which gives the system more flexibility to generate behaviour-specific code, and to optimize the read and write behaviour of shared data separately. Aurora can also return C-style pointers to shared data under controlled circumstances. The data in a PSR is always accessed using C-style pointers, which is efficient, but it does not allow the system to selectively intervene in data accesses. Lastly, Aurora supports multiple writers to the same distributed vector object, which can be important for performance [ACD+96], while PSRs only allow a single writer.

10 Concluding Remarks

Researchers have explored a variety of different implementation techniques for DSM and DSD systems. The Aurora DSD programming system is an example of a software-only implementation that uses data sharing optimizations to achieve good performance on a set of parallel programs.

What distinguishes Aurora from other DSM and DSD systems is its use of scoped behaviour as an interface to a set of data sharing optimizations. Scoped behaviour supports per-context and per-object flexibility in applying the optimizations. This novel level of flexibility is particularly useful for incrementally tuning multi-phase parallel programs and programs in which different shared objects are accessed in different ways. The performance of Aurora is encouraging and future work will explore new data sharing optimizations and how they can exploit different network performance characteristics.

Scoped behaviour can be implemented in standard C++ without special compiler support and it offers important safety benefits over typical run-time libraries. The technique appears to be a viable approach for supporting this form of optimization flexibility.

11 Acknowledgments

Thank you to Ben Gamsa, Eric Parsons, Karen Reid, Jonathan Schaeffer, Ken Sevcik, Michael Stumm, Greg Wilson, Songnian Zhou, and the anonymous referees for their comments and support during this work. Thank you to the Department of Computer Science and NSERC for financial support. Thank you to ITRC and IBM for their support of the POW Project.

References

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[AG96] S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[BCZ90] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. 1990 Conference on Principles and Practice of Parallel Programming*. ACM Press, 1990.

[BKT92] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[Boo91] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.

[BZS93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. 38th IEEE International Computer Conference (COMPCON Spring'93)*, pages 528–537, February 1993.

[Cop92] J.O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison–Wesley, 1992.

[DGLS93] N.E. Doss, W.D. Gropp, E. Lusk, and A. Skjellum. A Model Implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1993.

[FLR+94] B. Falsafi, A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proc. Supercomputing '94*, pages 380--389, November 1994.

[GLL+90] K. Gharachorloo, D.E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GLS94] W.D. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[JKW95] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. 1988 International Conference on Parallel Processing*, volume II, pages 94–101, August 1988.

[LLS+93] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19: 1079–1103, 1993.

[LQ92] M. Lemke and D. Quinlan. P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications. In *Proc. CONPAR 92–VAPP V*. Springer-Verlag, September 1992.

[Lu97] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997. Available at http://www.cs.utoronto.ca/~paullu/.

[OEPW96] W.G. O'Farrell, F.Ch. Eigler, S.D. Pullara, and G.V. Wilson. ABC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[Pth94] Draft Standard for Information Technology—Portable Operating Systems Interface (Posix), September 1994.

[SGZ93] H.S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence. In *Proc. Symposium on Principles and Practices of Parallel Programming*, pages 229–238, May 1993.

[SL94] D.J. Scales and M.S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. 1st Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[SS92] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[Sun96] Sun Microsystems. *The Java Language Specification, Version 1.0*, August 1996. http://www.javasoft.com/doc/language_specification/.

Extending the Standard Template Library for Parallelism in *Coir<Futures>*

Neelakantan Sundaresan
Applications Development Technology Institute
IBM Software Solutions Division
555 Bailey Avenue
San Jose, CA 95141

e-mail: neels@vnet.ibm.com

Abstract

The Standard Template Library (STL) [15, 19] is a C++ implementation of the generic programming paradigm [16]. Unlike in typical container class libraries, algorithms in this paradigm do not work directly on collection container objects. They work on iterators (access and traversal objects) exported by containers. Given N data types, M containers, and K algorithms as components of a software system, STL provides a mechanism – using C++ templates and the generic programming paradigm – to reduce the possibly $N * M * K$ implementations to $N + M + K$ implementations.

Over the last decade significant research has been done in the area of object-oriented parallelism and a number of models, libraries, and language extensions have been proposed and implemented. C++ has been an important language for writing parallel libraries and for extending for parallelism. In this paper we discuss how the generic programming paradigm in STL can be used and extended to support parallel programming in a manner that allows good expressibility, code reuse, and extensibility of the library. We look at control and data-parallel abstractions. We also discuss how different strategies for work distribution in parallel algorithms can be supported in the spirit of generic programming. We describe the relevant abstractions in *Coir<Futures>* [23], our STL-based parallel C++ library for shared memory parallelism.

1. Introduction

The object-oriented paradigm has been successful in providing good expressibility, maintainability, and reuse of software systems. With the growing popularity of the object-oriented paradigm, researchers in the parallel compiling and computing world have experimented with using this paradigm for parallelism. Using and extending C++ has been popular [25] though a few systems based on Eiffel [14] and Smalltalk [10] have also been built.

Generic programming [16] is a paradigm that abstracts concrete, efficient algorithms that can be combined with different data representations to produce a wide variety of useful software. For instance, using this paradigm, a generic sorting algorithm can be instantiated to work with different aggregate data structures like linked lists or arrays. Originally developed in Ada and Scheme, such a library has been recently implemented in C++ as the STL [19, 15] and in Java [17]. STL has been adopted by the C++ ANSI standard committee [1]. Its success is inevitable as C++ programmers have discovered a new style of writing container class libraries.

As practitioners of object-oriented parallel processing, we believe that the generic programming paradigm adds a new dimension to building parallel libraries by enabling better reuse of sequential code written in the same paradigm, by allowing the writing of extensible parallel programs, and providing interesting and useful parallel abstractions. The *Coir<Futures>* shared memory parallel system was designed with this belief. In the following paragraphs, we will see how abstractions for asynchronous parallelism and for data parallelism are supported in *Coir<Futures>* along the lines of the generic programming paradigm.

This paper is organized as follows: In the next section we discuss generic programming and STL. In section 3. we describe our *Coir<Futures>* library briefly. In section 4. we discuss control parallel ab-

stractions through futures [9] and in section 5. we describe data parallel objects. In section 6. we discuss data parallel extensions to sequential STL algorithms. In section 7. we discuss related work and in section 8. we draw our conclusions.

2. Generic Programming and the Standard Template Library

The generic programming paradigm prescribes four kinds of abstractions: data abstractions, algorithmic abstractions, structural abstractions, and representational abstractions.

Data abstractions are data types and sets of operations on them. C++ provides templates as the necessary constructs for data abstractions. Templates provide a uniform interface and implementation abstractions for different data types. For instance, a template stack class can be instantiated to a stack of integers, doubles, or any user-defined type. Thus, for N data types only one template container class is provided which can be instantiated N ways.

STL provides implementations for the remaining three abstractions through STL algorithms, iterators, and adaptors. These are discussed below.

Algorithms

Generic algorithmic abstractions are families of data abstractions with a common set of algorithms. In order to make algorithms generic they are designed to work on iterators (see below) that are exported by containers. For instance, a sort algorithm could work on a linked list or a vector data abstraction if the list and vector collection classes provide iterator objects that mark the beginning and end of the container. Algorithms are implemented as template functions in STL, typically parameterized over iterators or structural abstractions.

Iterators

Iterators are implementations of structural abstractions and are data type templates exported by container classes. Iterators are generalizations of array pointers for generic containers and provide operators to traverse the range of data they point to and also operators to reference the element they point to. Typically, the pointer arithmetic operators like ++ (auto-increment), −− (auto-decrement), $+n$ (jump n positions forward), and $-n$ (jump n positions backward), are overloaded to provide traversal implementations. They also overload the comparison operators (==, <, >, =) and the assignment operator (=) to compare iterator positions and allow iterator assignments, respectively. The C++ * operator is overloaded to reference the element at the position pointed to by the iterator. Algorithms work over iterators rather than directly over containers. Therefore the same algorithm can work for different container types as long as they export appropriate types of iterators. An additional advantage of having algorithms work on iterators instead of on containers is that algorithms can be used to work on a partial range of elements in a container.

An iterator can be of one of the following kinds - input, output, forward, bidirectional, or random-access. Input iterators are data sources (e.g. the *cin* standard input object in C++), and output operators are data sinks (e.g. *cout* in C++). Forward iterators satisfy properties of both input and output iterators. Forward iterators can be traversed one position at a time only in the forward direction, hence they support only the ++ operator. Bidirectional iterators satisfy properties of forward iterators, can be traversed in both forward and reverse directions one step at a time, and support the −− operator. Random access iterators are bidirectional iterators, which can make non-unit jumps in the forward or reverse direction. They support the $+n$ and $-n$ operators too.

Most container classes export member functions called *begin()* and *end()* which return iterators that point to the first element and past the last element, respectively, of the container object. Starting with these functions, and using the iterator traversal operators, users can construct iterators pointing to a subrange of the elements in a container.

Adaptors

Generic representational abstractions are mappings from one structural abstraction to another. Called adaptors in STL, these abstractions are casting wrappers that change the appearance of a container (building a stack from a list), or an iterator (converting a bidirectional iterator to a reverse iterator). STL also has adaptors to convert C++ I/O streams and arrays to STL-style containers.

Functors

STL also defines function objects (or functors) which are basically template function pointers wrapped in template classes. These classes provide a '()' operator which is used for invoking the func-

tion. STL also provides adaptors to convert normal C++ function pointers to function objects.

2.1. Example

STL defines a container template class called *vector*. It also provides an algorithm called *find* which takes in three arguments – two input iterators *first* and *last* indicating a range, and a value argument *val* – and returns an iterator pointing to the first position in the range in which the element matches the value *val*. The following code builds a vector of integer elements, finds the position of the first zero element in the vector, and then the position of the next zero element in the vector.

```
...
// create a vector with 10 elements
vector<int> v(10);
// add elements to the vector
v.push_back(1);
v.push_back(5);
...
// point to the first element of the vector
vector::iterator i1 = v.begin();
// the following assertion will be true
assert(*i1 == 1 && *(i1+1) == 5);
// point past the last element of the vector
vector::iterator i2 = v.end();
// find the first zero occurrence
vector::iterator first = find(i1, i2, 0);
if(first == i2)
  cout << "vector has no zeroes" << endl;
else {
// find the second zero occurrence
  vector::iterator second = find(first, i2, 0);
  if (second == i2)
    cout << "vector has one zero" << endl;
  else
    cout << "vector has two or more zeroes"
         << endl;
}
```

3. The *Coir<Futures>* System

The ***Coir<Futures>*** library is our STL-based generic parallel programming library for shared-memory systems. It is built on top of a light-weight user-level thread library called *Coir-Core* [22] which supports standard thread operations for shared memory machines, and, in addition, includes support for locality, affinity, and migration domains.

Coir<Futures> recognizes control parallelism and data parallelism as two important models of parallelism. Control parallel abstractions provide a mechanism for specifying that one piece of computation can proceed in parallel with and independent of another piece of computation. *Coir<Futures>* supports control parallelism through future abstractions which are built on top of thread constructs. It supports thread classes and their inheritance mechanism. It has support for monitor-style programming and it separates thread-level operations from processor-level operations.

Data parallelism is a powerful and most commonly provided and used construct for parallelism, especially in scientific parallel computing. In a data parallel model, multiple processors/threads perform a common computation but each processor/thread operates on different data. In languages that have functions as first class data types, functional data parallelism can be defined by representing the common computation by a function or a function pointer. Traditionally data parallelism has been provided as a processor-level abstraction. The most common style is SPMD (Single Program Multiple Data) parallelism [6] where each processor executes the same program but based on the processor id each one executes it on a different section of the data. The program computation is interspersed with synchronization and data exchange. *Coir<Futures>* supports functional data parallelism at the thread level through the abstractions of thread groups called *ropes* [21]. Ropes are powerful data parallel objects which provide a scoping mechanism for non-blocking, multithreaded, and interleaved data parallelism, deviating from the popular but restricted SPMD-style data parallelism. *Coir<Futures>* has template facilities to customize these objects based on the function signatures.

4. Futures for Control Parallelism

Futures [9] are place-holders or 'IOUs' for values or computations. Future abstractions are useful in parallel programming because they allow asynchronous computation by enabling delaying the computation of the value of an object until the value is required. Good abstractions for futures are useful in writing parallel programs that look similar to their sequential counterparts. *Coir<Futures>* supports two kinds of futures – data futures and computation futures. Data futures are simple place-holders for values. The futures are resolved by some

arbitrary computation entering a value into the future. Computation futures are those that represent a delayed computation (execution of a function). The value of a computation future is the value returned by the function that represents the computation. Data futures are templatized on the base types of the values they represent. Computation futures are templatized on the signature of the computations (functors) they represent. Since the return type of a function is a part of its signature, the base type of the value that the future represents is hidden in the template parameter. Future classes also support a cast operator to their base types. The cast operator is a blocking operator and waits for the future resolution i.e., for the future to have a valid value.

4.1. Computation Future Functor Adaptors

The programming paradigm of the STL goes very well with the future abstraction described above. For instance, future objects can be created out of arbitrary function objects. STL has adaptors to convert N-ary function pointers to N-ary function objects where N = 1, 2. Function objects and adapters for N = 3, 4, ... can be easily built. *Coir<Futures>* defines adaptors that convert N-ary function pointers to future N-ary function objects. This is done using the following two constructs [1]:

1. A *future_pointer_to_*N-ary*_function* template class for every N-ary function that provides the necessary type definitions and member functions for future-based operations.

2. A *future_ptr_fun* template function that is an adaptor from an N-ary function pointer to an instance of a *future_pointer_to_*N-ary*_function* class.

4.1.1. *future_pointer_to_*N-ary*_function*

This template class is defined as follows:

```
template <class Arg1, class Arg2, ..., class ArgN,
          class Result>
class future_pointer_to_N-ary_function
  : public N-ary_function<Arg1, Arg2, ..., ArgN,
                          Result >
{
protected:
  pointer_to_N-ary_function<Arg1, Arg2, ..., ArgN,
                            Result> Nf;
public:
  typedef TaskN<Arg1, Arg2, ..., ArgN,
                Result>::future_type future_type;
  future_pointer_to_N-ary_function(
    Result (*f)(Arg1, Arg2, ..., ArgN))
    : Nf(ptr_fun(f))
      { ... }
  ~future_pointer_to_N-ary_function()
     { ...clean up task lists... }
  future_type operator()(Arg1, Arg2, ..., ArgN);
};
```

This template class has $N + 1$ parameters – N parameters (*Arg1, ..., Arg*N) for the N argument types of the N-ary function this corresponds to, and the last parameter(class *Result*) for the result type of the N-ary function.

This class is defined as a subclass of the N-ary*_function* template class provided in STL. (Note that STL defines only unary and binary function template classes. Ternary, quaternary function template classes etc. can be easily defined.)

The template class defines/exports a public type definition – *future_type*. This is the type of the handle that is returned when a '()' operator of an object of this class is invoked (see below). The implementation of *future_type* should ensure that an object of *future_type* should be castable to *Result* type.

The constructor of the template class takes one argument which is an N-ary function pointer. It converts this N-ary function pointer into a *pointer_to_*N-ary*_function* object using the *ptr_fun* adaptor provided in STL and stores it in its member defined by N*f*.

The template class TaskN stands for a thread or a task implementation class that exports a future template data type. One requirement on this future data type is that it has a cast operator that can be used to convert it to *Result* type. In a typical implementation the future will have fields that mark the status of the underlying task and whether the future has been resolved. The cast operator will check this field to see if the task underlying the future has been scheduled and completed. If it is not completed it will block the current thread for the underlying to complete and then resolve the future to the value returned by the task. If the task is completed before this cast operator is invoked, the return value is saved and the future is marked to have a valid value. In this case, the cast operator returns immediately with the saved value as the

[1] We have freely used N-ary in the code fragments to stand for unary, binary, ternary, ... The actual system has separate constructs for each of these cases.

value of the future.

The *future_pointer_to_*N-ary*_function* class also supports a '()' operator which is basically an invocation operator of the future function object. It expects N arguments and creates a task of type TaskN with the N-ary function objects and these N arguments and returns a *future_type* object. The created task is added to an internal list of tasks. When the destructor of the *future_pointer_to_*N-ary*_function* object is invoked (when the object is deleted or goes out of scope), this internal list and the tasks in that list are deleted and disposed.

4.1.2. *future_ptr_fun*

The adaptor *future_ptr_fun* is used to convert an ordinary N-ary function pointer to a future function object as described above. It is defined as follows:

```
template <class Arg1, class Arg2, ..., class ArgN,
          class Result>
future_pointer_to_N-ary_function<
    Arg1, Arg2, ..., ArgN, Result>
  future_ptr_fun(Result (*x)(Arg1, Arg2, ...,
                             ArgN))
{
    return future_pointer_to_N-ary_function<
            Arg1, Arg2, ..., ArgN, Result>(x);
}
```

This template function also is parameterized on $N + 1$ parameters – N argument types and one result type. The function itself takes one argument which is an N-ary function pointer, builds a *future_pointer_to_*N-ary*_function* object (passing this function pointer to its constructor), and returns this object. Figure 1 shows how this adaptor behaves.

4.1.3. Example

The following piece of code is an example of how the user of this library can use this facility. This code shows how a unary function pointer can be 'adapted' to a future unary function object and used.

Suppose *int (*foo)(int)* is a unary function pointer that expects an integer argument and returns an integer result. It can be converted to a future unary function object and used as follows:

```
future_pointer_to_unary_function<
  int,int>::future_type fval
    = (future_ptr_fun(foo))(4); // (A)
```

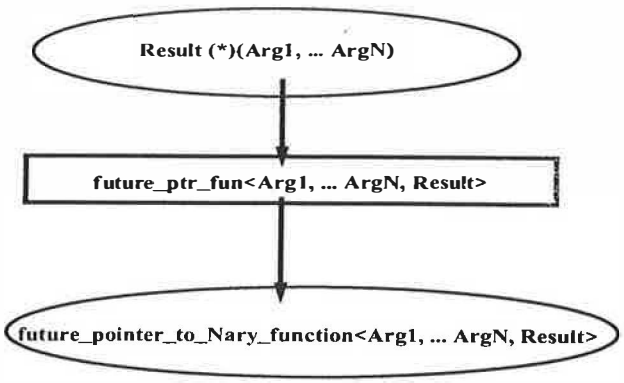


Figure 1: The Future adaptor function template behavior: It takes a function pointer for an N-ary function and returns a future N-ary function object.

```
// do some unrelated work      // (B)
int final = fval + 3;          // (C)
```

The code executes as follows (see figure 2 for a pictorial representation of the execution of this code):

In the statement labeled (A) the following sequence of operations takes place: The unary function pointer *foo* is converted into a future unary function object with the use of the adaptor *future_ptr_fun*. Then the operator '()' is invoked on the resultant future unary function object with argument *4*. This results in a task being scheduled to compute *foo* with argument *4* and a future handle is returned.

In the statement(s) labeled (B) the user code can perform computations that do not need the results of the *foo* function described above.

In the statement labeled (C) the result of the *foo* function is required and the user code adds *3* to this result and assigns it to *final*. Note that *fval* is of a future type while *3* is of type *int*. Since *+* is only defined between similar types (and not between future and *int* types) this statement would be valid only if there was a way to go from the future type to its *Result* type (i.e., *int* type). As mentioned before, every future type has to provide a cast operator to its *Result* type. In the implementation of the cast operator, the current thread waits for the task corresponding to this future resolution to finish and returns the value returned by the task.

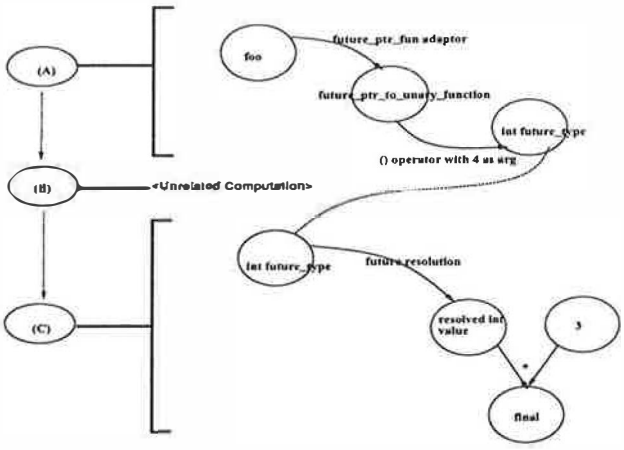


Figure 2: The execution pattern of the example given in page 5: The statement labeled (A) creates an *int* future object out of the *foo* function pointer by first creating a *future_ptr_to_unary_function* object using the *future_ptr_fun* adaptor and schedules it with argument 4. Statement (B) represents an unrelated computation. Statement (C) requires the future to be resolved as a result of the addition operation between the *int* future object and an integer constant 3 to produce 'final'.

4.2. Future-based Algorithms

Typically parallel data abstractions are built around arrays or collection types. Since multiple processors and address spaces may be involved, the abstractions are for their distribution, access, and update across memory and processor spaces. Iterators provide a way to traverse aggregate data types and to reference and update particular elements. STL algorithms work using iterators (instead of working on containers directly), iterators are provided by containers, and containers are template classes. Just as containers are instantiated on regular C++ data types, they can be instantiated on future type instantiations. Because iterator traversal and access operators do not assume anything about the data type contained in the containers, the same programs that work for containers of basic data types can work for containers of future data types. Asynchronous and delayed computations come for free, as values are not required until the point where the dereferencing * iterator operator is used to access the value of the base type. When the result type of the use of the * operator resolves into a base type, future resolution takes place. Thus sequential STL applications can be easily adapted to future-based parallel applications. Future-based iterators and containers can be assigned, copied, and passed as arguments to the point where their values are accessed. This allows us to write programs where delayed evaluations can be exploited across procedure boundaries.

4.2.1. Example

The STL *find* algorithm looks for the first element in a list with value = *value*. It is written as follows:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                   InputIterator last,
                   const T& value)
{
   while (first != last && *first != value)
   {
      ++first;
   }
   return first;
}
```

The following code fragment builds a list of future computations and executes *find()* over this list to find the first element in the list with a zero value.

```
int foo(const int i);
typedef Future<int> future_type;
typedef list<future_type> list_type;
list_type xlist;
for(int i = 0; i < N; i++)
   xlist.push_back(future_type(foo, i)); // add to list
...
list_type::iterator pos
  = find(xlist.begin(), xlist.end(), 0);
```

Here *find* computes only to the point where it finds an element with a value = *value*. The remaining futures need not even be computed or resolved.

4.3. Future Implementations

Currently *Coir<Futures>* implements futures using an extended form of the leapfrogging mechanism discussed in [24]. This mechanism combines leapfrogging with task stealing to achieve load balancing. Since this paper concentrates more on the generic programming interface aspect of the system, we do not describe the implementation details. It may be noted that due to the object-oriented nature

of the system different implementations (depending on the underlying thread system and the hardware environment) may be supported without changing the interface.

There may be situations (like the one described in the example in the previous section) where future objects are created but are never resolved. The semantics and our implementation guarantees that the future value is made available at the point where it is required. However, futures may be scheduled and computed even if their values are not required. Consider the following example:

```
future_pointer_to_unary_function<
  int,int>::future_type fval1
    = (future_ptr_fun(foo))(4);
future_pointer_to_unary_function<
  int,int>::future_type fval2
    = (future_ptr_fun(foo))(5);
...
int x = fval1;
...fval2 is never resolved...
...x is never used...
```

This program creates two future computations *fval1* and *fval2*. *fval2* is never resolved while *fval1* is resolved to an integer variable *x*, but the resolved variable *x* is never used. Most optimizing C++ compilers which performs extensive flow analysis may be able to eliminate dead code of this kind. In compiler terms, *x* and *fval2* are dead variables and their computation may be eliminated if other conditions like absence of side effects in their computation are satisfied. *Coir<Futures>* does not make any special effort to avoid scheduling dead futures. We rely on the C++ compiler for such optimizations. The compiler may not be able to eliminate all cases of dead futures. For instance, the ones in the example in the previous section are embedded in a list and are harder for the compiler to identify.

5. Data Parallel Objects

In our future based environment, data parallel objects are schedulable objects not restricted to SPMD programming. Further, multiple data parallel computations can interleave. These data parallel objects can be thought of as future objects used to schedule data parallel operations where each thread operates on a different piece of data. The future resolution is a reduction of the values returned by each of the parallel functions. Just as the underlying control parallel component in *Coir<Futures>* is a thread, the underlying data parallel component is a rope or a group of threads. Rope objects define functional data parallelism and synchronization among threads. A global rope is defined from the main threads of all the processors participating in a parallel program. The following are some of the functions provided by the base Rope class:

1. static Rope& SelfRope() - identify the currently executing rope
2. int Size() const - number of threads in a rope
3. int SelfIndex() const - index of the currently executing thread in its rope
4. int Index(const Thread& thr) const - index of the thread 'thr' in the rope
5. Thread& operator[](const int index) const - the 'index'th thread in the rope.
6. Reduction& ReductionObj() const - the reduction operation skeleton in the rope (see below for description.)

Derived rope classes can be templatized on type signatures of the the data-parallel tasks they would execute. *Coir<Futures>* defines *UnaryRope* and *BinaryRope* derived template classes where the template parameters specify that the tasks are represented by unary and binary function object types, respectively.

5.1. Reduction

An important aspect of data parallelism is the reduction operation where each thread contributes a value and the values are reduced using a function to obtain and return a reduced value to each of the threads. Reduction can be defined follows:
Let S_τ be the set of all elements of type τ. Let $\wp(S_\tau)$ denote the set of all finite subsets of S_τ. Let $S^\tau_\otimes$ be the set of all commutative and associative binary operators/functions defined over elements of type τ. Then a reduction operator, ρ, is defined from $S^\tau_\otimes \times \wp(S_\tau) \longrightarrow S_\tau$. A reduction operation is the application of an operator $\otimes \in S^\tau_\otimes$ to a set of elements of type τ and returns a result of type τ. Operationally, if S_τ is the set $\{a_1, a_2, a_3, \ldots, a_n\}$, then

$$\rho(\otimes, S_\tau) = \otimes(a_1, \otimes(a_2, \otimes(a_3, \ldots \otimes (a_{n-1}, a_n))))$$

is one way of applying the operator.

As an example, let τ be the *int* type, and S_τ be instantiated to a vector v of type $v <$

$int >$ with 10 elements $\{4, 5, 7, 3, 2, 10, 5, 8, 0, 12\}$. Let $\otimes$ be the addition operator $+$. Since $+$ is associative over the set of integers, it qualifies as a reduction operation and $\rho(+, v) = +(4, +(5, +(7, +(3, \ldots, +(0, 12))\ldots))) = 56$.

The reduction operation itself is a parallel operation that can be done with $O(logN)$ complexity using a tree-style operation. There are three aspects to reduction:

1. The data types of the individual values contributed by each of the threads.

2. The computation pattern of the reduction operation.

3. The reduction operator or function that is used to reduce the values. This should be an associative function (i.e., f(f(x,y),z) = f(x, f(y,z))).

It may be noted that the reduction computation pattern is independent of the data types of the values contributed. It just depends on the amount of parallelism that is available in terms of the number of processors and threads. Tree reductions are effective in reducing the parallel complexity. However, doing the reduction operation this way in parallel may demand that the reduction operator not only be associative but also be commutative (i.e., f(x,y) = f(y,x)).

Since the computation pattern is independent of the data types of the values and the reduction operator, a computation pattern skeleton can be built at the time of rope creation. This skeleton can be used and reused for different reduction operations. The way *Coir<Futures>* defines the computation pattern, it has the following properties:

1. Reduction operations are rope-specific. Thus reduction operations belonging to different ropes are non interfering.

2. Two different reduction operations within the same rope are non-interfering. This is ensured by defining this skeleton to consist of two trees - a fan-in tree and a fan-out tree.

5.1.1. The Reduction Skeleton

The reduction skeleton consists of a fan-in tree and a fan-out tree.

The fan-in tree has N nodes, where N is the number of threads in the rope. Each node is identified by a distinct thread index (0 to $N-1$). During the fan-in reduction computation the reduction operation takes place in a bottom up fashion - starting at the leaf and going to the root. At the end of this the root has the reduced value.

The fan-out tree also has N nodes and the nodes are identified by thread indices. The fan-out phase is a broadcast phase where the reduced value is broadcast to each individual thread in a top-down fashion - starting at the root and going to the leaf.

In reality there is only one tree as specified in the Reduction class, only the traversals specified by fan-in and fan-out are different.

The Reduction Class

The Reduction tree class description is parameterized so that it will work for any tree which is unary to (N-1)-ary [2], and also so that the fan-in and the fan-out trees are of different ranks (i.e., one can have a binary fan-in tree and a quaternary fan-out tree.). Past research has shown that the ranks of the fan-in and fan-out tree have a significant effect on parallel synchronization performance, and should be determined based on the architecture, number of processors, and memory hierarchy of the parallel system [13].

```
template <class SizeType>
struct FanInNode
{
  typedef SizeType fanin_size_type;
  enum { fanin_size = sizeof(fanin_size_type) };
  bool ith_fanin_child_exists(const int i);
  ...
};

template <class SizeType>
struct FanOutNode
{
  typedef SizeType fanout_size_type;
  enum { fanout_size = sizeof(fanout_size_type) };
  bool ith_fanout_child_exists(const int i);
  ...
};

struct Reduction
{
  struct MyNode : public FanInNode<short>,
                  public FanOutNode<int>
  {
    ...
  };
  typedef MyNode node_type;
```

[2] read (N-1)-ary as (N minus 1)-ary

```
  Reduction(const int size); // constructor
  virtual ~Reduction(); // destructor

  // number of threads participating
  int Size() const;
  // get the node corresp. to the ith thread
  node_type& get_node(const int i);
  // get this thread's node
  node_type& get_my_node();
  // index of the fanin parent of the ith thread
  static int fanin_parent(const int i);
  // index of the fanout parent of the ith thread
  static int fanout_parent(const int i);
  // index of the kth fanin child of the ith thread
  static int fanout_child(const int i, const int k);
  // index of the kth fanout child of the ith thread
  static int fanin_child(const int i, const int k);
};
```

Implementation Details

The class *FanInNode* defines the fan-in node properties. It defines a type *fanin_size_type*. For a fan-in size of k this is a C++ data type whose size (as given by the C++ *sizeof* operator) is k bytes. For instance, the code defines *fanin_size_type* to be a short to specify a fan-in size of 2 (given by the field *fanin_size*), since *short* is 2 bytes in many implementations [3] This class also defines a boolean query member function, which when given the index i says whether or not it has an ith child (note that any node in the fan-in tree has at most *fanin_size* children).

The class *FanOutNode* defines the fan-out node properties. It defines a type *fanout_size_type*. For a fan-out size of k this is a C++ data type whose size (as given by the C++ *sizeof* operator) is k bytes. For instance, the code defines *fanout_size_type* to be an *int* to specify a fan-out size of 4 (given by the field *fanin_size*), since *int* is 4 bytes in many implementations. This class also defines a boolean query member function, which when given the index i says whether or not it has an ith child (note that any node in the fan-out tree has at most *fanout_size* children).

The four functions *fanin_parent*, *fanout_parent*, *fanin_child*, and *fanout_child* are *static* because these functions depend only on the *fanin_size* and *fanout_size* quantities which are constants and the arguments passed to these functions. This permits the C++ compiler to perform inlining optimizations.

[3] It may seem that making assumptions about sizes of *ints* and *shorts* is against the object-oriented paradigm, it is important for obtaining good performance. *ints* and *shorts* can be used in conditional expressions and can be checked using a single scalar compare operator. Since the fields on which the condition checks are made are prone to shared-memory bottlenecks such optimizations contribute to significant performance improvements.

5.1.2. Type-specific Reduction

In a data parallel operation involving N threads of a rope, the threads can participate in a type-specific reduction. As discussed above each rope object has the skeleton of a tree-based reduction operation. The threads in a rope can enter type-specific reductions by cloning this reduction skeleton to a reduction object for that type. To achieve this *Coir<Futures>* defines a *ReductionT* template class parameterized on the type of the value the threads contribute to a reduction operation. This class is defined as follows [4]:

```
template <class T, class ReducerType
                    = binary_function<T,T, T> >
class ReductionT
{
public:
  typedef ReducerType reducer_type;
  typedef T data_type;
  ReductionT(Reduction& my_red
    = Rope::SelfRope().ReductionObj());
  ~ReductionT();
  // reduction operator
  T operator()(reducer_type reducer,
               const T& data);
};
```

5.2. Using Reduction Objects

The user code uses the reduction facility as given below. Each thread does the following:

1. Obtain the reduction tree skeleton object corresponding to this thread's rope.

2. For each type T for which reduction operation is to be performed, create a type-specific per-thread reduction object.

[4] The declaration shown here uses template parameter constraint where ReducerType is specified to be of type binary_function. This is only for documentation purposes. Our compiler did not support specifying template constraints at the time of this implementation.

3. For each reducer (binary commutative and associative operator for type *T*), invoke the '()' operator of the ReductionT object.

The following piece of code shows a sum reduction followed by a product reduction on integer types. It is executed by each thread in the rope.

```
// build the type specific reduction object
ReductionT<int, binary_function<int, int, int> >
   red_obj(Rope::SelfRope().ReductionObj());
int my_contrib = ...
// sum reduction
int my_sum = red_obj(plus<int>, my_contrib);
// product reduction
int my_prod = red_obj(times<int>, my_contrib);
```

where *ReducerType* is a type of a binary function object which expects two arguments of type convertible to type *T*, and result type is a type convertible to type *T*. The constructor expects a *Reduction* object as an argument which is typically *Rope::SelfRope().ReductionObj()*. The class also exports a type called *reducer_type* which is the same as the actual argument for the template parameter *ReducerType*, and a type *data_type* which is basically type *T*. The *()* operator takes two arguments - *reducer* is a binary commutative and associative function object that is used in the reduction, and *data* is the contribution of the thread to the reduction operation. The *()* operator performs the actual reduction - each thread participating in the reduction operation invokes the *()* operator while in a data parallel computation. Note that all the threads should specify the same *reducer_type* argument. Making it a part of the *()* operator allows us to reuse the same reduction object for a different reduction operation easily.

6. Data Parallel Algorithms

STL-provided algorithms and typical user-written algorithms use one or more iterators as arguments and traverse, build, and update containers using these iterators. In providing parallel implementations of these algorithms it would be useful to design the implementations in such a way that different strategies of work distribution can be made a part of the algorithm implementation. In the sequential world, STL reduces N*M*K possible implementations for N data types, M containers and K algorithms to N+M+K implementations. In the parallel world, given P strategies of work distribution, we would like to extend STL so that N*M*K*P implementations are reduced to N+M+K+P implementations. This can be done with the support of per-thread parallel iterators and strategy classes discussed below.

Data parallel versions of sequential algorithms have a group of threads participating in the algorithm. The parallel implementation of a sequential algorithm is a template function with one template parameter in addition to its sequential counterpart. This parameter represents the *Strategy* class. The parallel algorithm function accepts an extra parameter as compared to the sequential counterpart. This parameter is the strategy object which is an instance of the Strategy class. In the body of the parallel algorithm, the strategy object is used to convert each of the sequential iterators to per-thread parallel iterators. The per-thread parallel iterator traverses the container in such a way that it touches those parts of the container for which the thread that this iterator corresponds to is responsible. When each thread has computed a partial result the results are composed through a reduction operation.(See figure 4 and compare with figure 3). (Note that all STL algorithms cannot be written exactly this way. Mainly because the composition of the partial results may not be an associative or commutative operation for a particular style of work distribution.)

6.1. Per-thread Parallel Iterators

STL iterators are inherently sequential in nature because each iterator defines a single cursor of traversal and update. This is inadequate for parallel programming. We can define per-thread iterators based upon the strategy used for accessing and traversing the iterator space by the threads taking part in a data parallel operation.

6.2. Strategies

Strategies are data types that specify how work is distributed over the threads participating in a data parallel algorithm. As algorithms typically operate over iterators, strategy classes convert sequential iterators to per-thread parallel iterators. Conceptually, strategies define iterator adaptors. A strategy class is templatized on iterator types, value types, reference types, and distance types. It supports operators that, given sequential iterators, return per-thread parallel iterators. Strategies may be static or dynamic.

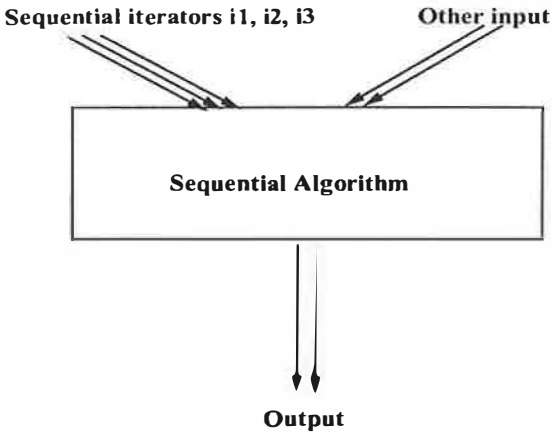


Figure 3: Typical forms of sequential algorithms in STL style: The sequential algorithm takes in some iterators (the one in the figure takes 3) and other input parameters, traverses the iterators sequentially and produces an output.

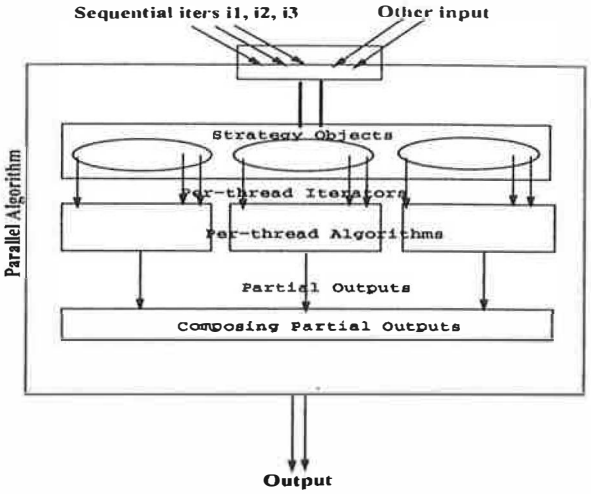


Figure 4: In a typical parallel algorithm strategy objects are used to convert the sequential iterators to per-thread parallel iterators which are used in the partial algorithms by each thread. The results computed by each of the threads in the partial algorithms are composed to produce the final output.

Static Strategies

Static strategies specify 'compile-time' work distribution where the region of the iterator space that a thread works on is decided based on the thread index, number of threads, and the size of the iteration space. *Block* and *Cyclic* strategies are examples of this strategy type. Given an iteration space with N values, with indices 0 to $N-1$, over which T threads operate, the iteration space of the ith thread in the *Block* strategy is given by $\{ \frac{Ni}{T}, \frac{Ni}{T}+1, \ldots, \frac{N(i+1)}{T}-1\}$ where $0 \leq i < T$. The iteration of the ith thread in the Cyclic strategy is given by $\{ i, i+T, i+2T, \ldots, i+(\frac{N}{T}-1)*T\}$.

Dynamic Strategies

Dynamic distribution strategies can be used for dynamic load-balancing. Here the traversal pattern of the per-thread iterators is decided dynamically. An example is the *Grab* strategy. Here all the threads in the rope operate over the original iterator space; the iterator traversal is monitor-based; only one thread can be doing an iterator operation at any time; and this operation affects any subsequent iterator operation by any other thread. Traversal of the iteration space is determined by the work load on the threads and the processors on which they execute and may be different between runs of the same program.

Figure 5 shows an example of the sequential traversal and parallel traversal in Block, Cyclic, and Grab strategies.

6.2.1. Requirements of a Strategy Class

The strategy class should export a type definition for a corresponding per-thread iterator for one or more STL sequential iterators. Providing a per-thread iterator for each of the sequential iterators makes a strategy class more usable in the sense that it can possibly be used to parallelize all STL-based sequential algorithms.

Many algorithms (e.g., *for_each*, *count*, *find*) in STL work with iterator pairs - to indicate the beginning and end of a sequence. They might also have a third iterator that follows the traversal between the first two iterators. To help the parallelization of such algorithms a strategy class may provide two '()' operators:

1. *operator()(const int size, iterator_type begin, iterator_type end, thread_iterator_type& thr_begin, thread_iterator_type& thr_end);* This operator basically returns the per-thread iterators given a pair of sequential iterators. The size parameter indicates the number of participating threads. This operator may be defined for one or more of the iterator types.

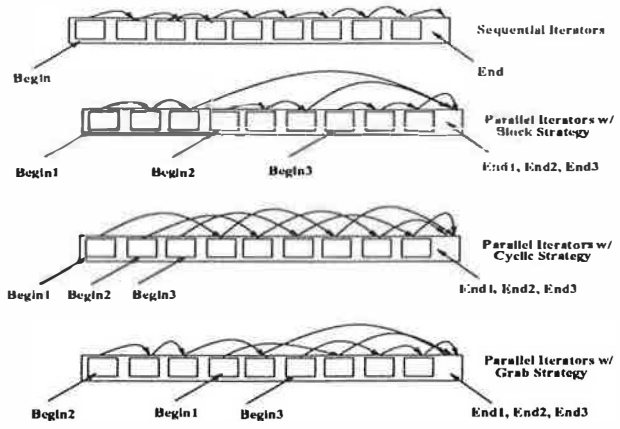


Figure 5: Sequential iterators and the corresponding per-thread parallel iterators with 3 threads for Block, Cyclic, and Grab Strategies: For each of *Begin* and *End* sequential iterators there are 3 per-thread iterators. Their traversal pattern depends on the strategy used.

2. *operator()(const int size, iterator_type1 begin1, iterator_type1 end1, iterator_type2 begin2, thread_iterator_type1& thr_begin1, thread_iterator_type1& thr_end1, thread_iterator_type2& thr_begin2);* In addition to returning per-thread iterators corresponding to 'begin1' and 'end1', this also returns the per-thread iterator corresponding to iterator 'begin2' that depends on 'begin1' and 'end1'.

In many algorithms the traversals from *begin1* and *begin2* typically happen in sync and make some assumptions about the relative positions. So the corresponding per-thread iterator should not lose this relation information. Static strategies can support iterators that retain this relationship because the traversal paths of the per-thread iterators are known. But dynamic strategies like the *Grab* strategy cannot guarantee this because the scope of the monitors used to control the traversal are per-iterator and using monitors over two iterator traversals would mean that we cannot reuse the sequential algorithm. Hence, while the parallel versions of some algorithms like *swap_ranges* and *transform* for static strategies are simple extensions of the sequential ones, the ones for the dynamic strategies may be quite different.

There are other algorithms in which even different static strategies might need different implementations. This depends on how the per-thread partial results are composed to produce the final results. For instance, the *remove* algorithm which removes elements of a particular value from a list for a *Cyclic* work strategy cannot compose the updated list by merely appending them in a reduction operation since this would put back the elements out of order. However, this can be done for the *Block* strategy.

Thus, it may not be possible to provide the same algorithm implementation for two different strategy classes for efficiency or correctness reasons. For this reason we introduce strategy tags and strategy categories similar to iterator tags and iterator categories in STL.

6.2.2. Strategy Categories and Tags

Coir<Futures> strategy classes are derived from two base classes - *static_strategy* and *dynamic_strategy*. These two classes are empty classes (similar to *input_iterator*, *output_iterator*, *etc.*, classes). Also we define a strategy tag corresponding to each strategy class (similar to iterator tags in STL) which can be used to provide implementations specific to different strategies. The strategy tag for any strategy class *S* can be obtained from the *strategy_category* function:

```
S_category_tag strategy_category(const S& strategy)
{
    return S_strategy_tag();
}
```

If an algorithm, say *transpose*, has different implementations for the *Block* and *Cyclic* strategies (either for the reasons of efficiency or for correctness) it can be implemented as:

```
template <class Strategy, class InputIterator>
OutputIterator transpose(Strategy& strategy,
                         InputIterator first,
                         InputIterator last)
{
  return transpose(strategy, first, last,
                   strategy_category(strategy));
}

template <class Strategy, class InputIterator>
OutputIterator transpose(Strategy& strategy,
                         InputIterator first,
                         InputIterator last,
                         block_strategy_tag btag)
{
```

```
    ... code specific to Block strategy ...
}

template <class Strategy, class InputIterator>
OutputIterator transpose(Strategy& strategy,
                         InputIterator first,
                         InputIterator last,
                         cyclic_strategy_tag ctag)
{
    ... code specific to Cyclic strategy ...
}
```

6.3. Example of a Parallel Algorithm

STL provides a *count* algorithm for counting the number of elements in an iterator range *first* and *last* that have value = *value*, and adds the result to the argument *n*. The following piece of code shows a sequential implementation:

```
template <class InputIterator, class T, class Size>
void count(InputIterator first,
           InputIterator last,
           const T& value,
           Size& n)
{
  while (first != last)
  {
    if(*first++ == value)
      ++n;
  }
}
```

The following algorithm provides a parallel implementation of the same using strategy classes. This assumes that each thread participating in a data-parallel operation invokes this function and provides its own local copy of *n*. The implementation uses the sequential version of *count* where each thread computes the partial results.

```
template <class Strategy, class InputIterator,
          class T, class Size>
void count(Strategy& strategy,
           InputIterator first,
           InputIterator last,
           const T& value,
           Size& n)
{
  //get the current data parallel object
  Rope& self_rope = Rope::SelfRope();
  //get the rope size
  int rope_size = self_rope.Size();
  //create a type-specific reduction object
  ReductionT<Size, plus<Size> >
    red(self_rope.ReductionObj());
  //create thread-specific iterators
  Strategy::thread_iterator my_first(first);
  Strategy::thread_iterator my_last(last);
  // position the iterators based on strategy
  strategy(rope_size, first, last, my_first, my_last);
  Size my_n = 0;
  // per-thread sequential algorithm
  ::count(my_first, my_last, value_pred, my_n);
  // combine the partial results
  Size total = red(plus<Size>(), my_n);
  n += total;
}
```

7. Related Work

One of the early works in C++ related to task parallelism is the AT&T task library [20]. Early thread libraries in C++ include Presto [4] and the Brown thread library [7]. While these systems are shared memory implementations, the ACE system [18] is a pattern based [8, 5] C++ implementation for distributed systems. A recent compilation of articles [25] discusses a number of parallel C++ systems. Most of the systems discussed in the collection extend the C++ language to support parallelism. ABC++ and the Amelia vector library are two systems which provide parallelism within C++. ABC++ has the future abstraction but it does not take the generic programming paradigm approach. The Amelia vector library uses an STL-based mechanism to support reduction operations. Combining future abstractions and multi-threaded data parallel mechanisms with STL is novel to our work. The only other parallel library implementation that incorporates STL is the HPC++ implementation from Indiana University [3]. HPC++ does not support future mechanisms, but has a definition for data parallel algorithms using parallel iterators for an SPMD model and has a distributed memory flavor. Though the interfaces and the architecture focus of HPC++ and *Coir<Futures>* are different, the parallel iterators defined in HPC++ can be compared to our strategy classes and the iterators defined by the classes. Since SPMD programs typically assume the "owner computes" rule, the HPC++ parallel iterators are designed around data distributions, while - since we have an underly-

ing MIMD paradigm and a shared memory architecture - our parallel iterators are designed around task distributions. Because massively parallel machines like the IBM SP/x are being designed using SMP nodes connected over distributed memory spaces, our abstractions can form the node component of the proposed HPC++ abstractions.

8. Conclusions

In this article we discussed how the generic programming paradigm can be used for parallel programming. The discussion was in the context of *Coir<Futures>*, our multi-threaded parallel C++ library. We introduced two abstractions - one for future-based control parallelism and the other for data parallel generic algorithms. The idea of futures is not new [2, 11, 24] but the notion of combining them with generic programming paradigms is one of the contributions of our work. Futures fit well with the template-based programming style of STL. Data parallel generic algorithms help reuse the corresponding sequential algorithms, provide an easy pattern for writing the parallel counterparts of the sequential algorithms, and provide a uniform interface and even implementations for different work distribution strategies in the spirit of generic programming. Generic reduction mechanism, and generic data-parallel algorithm design are some of the other contributions of this paper.

The idea of futures can be extended beyond just delayed computation. As futures are place-holders, they are similar to proxies. Futures need not only represent computation in the current address space, but can also be used for remote method invocations and persistent data types. The use of object-oriented and generic programming paradigms make it possible to provide a uniform interface for these objects with different behaviors and allow such objects to interact easily.

As we continue to work on the *Coir<Futures>* system, we are also working on more general ideas for software patterns [8, 5] for object-oriented parallel programming. Though some language-specific work already exists [12], specifying them beyond the language barrier will aid in interoperability.

References

[1] Working Paper for Draft Proposed International Standard for Information Systems Programming Language C++. Doc. No. X3J16/95-0185 - WG21/N0785, September 1995.

[2] E Arjomandi, W O'Farrell, I Kalas, F Eigler, and G Gao. ABC++: Concurrency by Inheritance in C++. *IBM Systems Journal*, 34(1):120–137, January 1995.

[3] Peter Beckman, Dennis Gannon, and Elizabeth Johnson. Portable Parallel Programming in HPC++. In *Proceedings of the International Conference on Parallel Processing (Software)*. CRC Press, August 1996.

[4] Brian Bershad, Edward Lazowska, and Henry Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software - Practice and Experience*, 18(8):713–732, August 1988.

[5] James Coplien, John Vlissides, and N Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, Reading, MA, 1995.

[6] F Darema, D George, V Norton, and G Pfister. A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[7] Tom Doeppner, Jr. and Alan Gebele. C++ on a Parallel Machine. Technical Report CS-87-26, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912-1910, November 1987.

[8] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable of Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[9] Robert Halstead, JR. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.

[10] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–109. ACM Press, 1989.

[11] Greg Lavender and Dennis Kafura. A Polymorphic Future and First-Class Function Type for Concurrent Object-Oriented Programming. To appear in the Journal of Object-Oriented Systems, 1995.

[12] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Java series. Addison Wesley, New York, 1996.

[13] John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[14] Stephen Murer, Jerome Feldman, and Chu-Cheow Lim. pSather Monitors: Layered Extensions to Object-Oriented Language for Efficient Parallel Computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA., June 1993.

[15] David Musser and Atul Saini. *STL Tutorial and Reference Guide, C++ Programming with the Standard Template Library.* Addison Wesley, 1996.

[16] David Musser and Alexander Stephanov. Generic Programming. In P Gianni, editor, *ISAAC '88, Symbolic and Algebraic Computation Proceedings*, Lecture Notes in Computer Science. Springer Verlag, 1988.

[17] ObjectSpace. The *Java™* Generic Library: JGL 1.0 User Guide. Available from the website http://www.objectspace.com, 1996.

[18] Douglas Schmidt. An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit. Technical Report WUCS-95-31, Washington University, St Louis, September 1995.

[19] Alexander Stephanov and Meng Lee. The Standard Template Library. Technical report, Hewlett Packard Laboratories, 1501, Page Mill Road, Palo Alto, CA 94304, December 1994.

[20] Bjarne Stroustrup and Jonathan Shapiro. AT&T Task Library. In *AT&T C++ Language System Library Manual.* AT&T, 1991. Part number: 800-5148-10, Revision A of February 1991.

[21] Neelakantan Sundaresan. *Modeling Control and Dynamic Data-Parallelism in Object-Oriented Languages.* PhD thesis, Computer Science Department, Indiana University, 215 Lindley Hall Bloomington, IN 47405, September 1995.

[22] Neelakantan Sundaresan. *Coir-Core: A Thread Library for Shared-Memory Parallelism.* Technical report, I.B.M., App. Dev. Tech. Institute, 555, Bailey Avenue, San Jose, CA 95141, October 1996.

[23] Neelakantan Sundaresan. Generic Parallel Programming in *Coir<Futures>*. Technical report, I.B.M., App. Dev. Tech. Institute, 555, Bailey Avenue, San Jose, CA 95141, April 1997.

[24] David Wagner and Brian Calder. Leapfrogging: A Portable Technique for Implementing Efficient Futures. *ACM Sigplan Notices: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 28(7):208–217, July 1993.

[25] Gregory Wilson and Paul Lu, editors. *Parallel Programming using C++.* MIT Press, 1996.

A Tool for Constructing Safe Extensible C++ Systems

Christopher Small
Harvard University

Abstract

The boundary between application and system is becoming increasingly permeable. Extensible applications, such as web browsers, database systems, and operating systems, demonstrate the value of allowing end-users to extend and modify the behavior of what was formerly considered to be a static, inviolate system. Unfortunately, flexibility often comes with a cost: systems unprotected from misbehaved end-user extensions are fragile and prone to instability.

Object-oriented programming models are a good fit for the development of this kind of system. An extensions can be designed as a refinement of an existing class, and loaded into a running system. In our model, when code is downloaded into the system, it is used to replace a virtual function on an existing C++ object. Because our tool is source-language neutral, it can be used to build safe extensible systems written in other languages as well.

There are three methods commonly used to make end-user extensions safe: restrict the extension language (e.g., Java), interpret the extension language (e.g., Tcl), or combine run-time checks with a trusted environment. The third technique is the one discussed here; it offers the twin benefits of the flexibility to implement extensions in an unsafe language, such as C++, and the performance of compiled code.

MiSFIT, the Minimal i386 Software Fault Isolation Tool, can be used as the central component of a tool set for building safe extensible systems in C++. MiSFIT transforms C++ code, compiled by g++, into safe binary code. Combined with a runtime support library, the overhead of MiSFIT is an order of magnitude lower than the overhead of interpreted Java, and permits safe extensible systems to be written in C++.

1 Introduction

Software fault isolation is a technique for transforming code written in an otherwise unsafe language (e.g., C or C++) into safe compiled code. At transformation time, each read, write, and jump instruction is analyzed and, if necessary, transformed to ensure that it will not reach outside the memory region assigned to the code.

Two other techniques for ensuring the safety of code are *safe languages* and *interpreted systems*. Safe languages, such as Java and Modula-3, are designed to make it difficult or impossible to write code that performs illegal or unsafe operations. By definition, safe languages are restricted; C++, which allows unchecked array accesses, pointer arithmetic, and arbitrary casting, is implicitly unsafe.

Scripting languages, such as Tcl and Perl, enforce safety by validating each data access as it takes place. Although great strides are being made to improve the performance of interpreted languages through the use of dynamic code generation [Hölze94], the performance overhead is at least a factor of two to ten over native compiled code.

In earlier work [Small96], we measured byte-code interpreted Java taking ten to seventy times longer than compiled C code performing the same task[1]. The overhead of software fault isolation is an order of magnitude less than that of interpretation, and SFI techniques have the advantage of operating on assembler-level code, so they can be used with any source language.

Although a small number of software fault isolation tools exist, and the underlying techniques are not complex, no tools have been made freely available on commodity platforms such as the x86. MiSFIT, the Minimal i386 Software Fault Isolation Tool, developed for use with the VINO extensible operating system, is such a tool. VINO is a new operating system, written in C++, designed around the idea that system policies can be modified, and kernel components reused, by downloading extensions written by untrusted end-users and protected by MiSFIT.

MiSFIT includes runtime support necessary to create a *sandbox* in which the downloaded code will run. Additional code (not provided as part of MiSFIT) is needed to load the extension into the base system, verify

This work was supported by a grant from Sun Microsystems Laboratories and by the John Parker Scholarship Fund.

1. The tests run in that paper were re-run with MiSFIT for this paper. The results are found in Section 6.1.

that the code was processed by MiSFIT, and offer a library of routines that can be called by the extension.

MiSFIT accepts x86 assembler code, produced by the Gnu C++ compiler, as input, and produces fault-isolated x86 assembler code as output. MiSFIT can be used as a component of a safe code system, allowing otherwise untrusted code to be linked to and run in the context of an extensible application or system. For example, MiSFIT can fault isolate dynamically linked extensions to world-wide web browsers (e.g., Netscape Navigator), kernel extensions (which are supported by a variety of current systems, such as Solaris, NetBSD, MS-DOS and Windows/NT), and client code linked to a database server (e.g., the Illustra database server [Bloor96]).

Software fault isolation techniques can be implemented in a compiler pass [Silver96], a filter between the compiler and assembler (as in the case of MiSFIT), or a binary editing tool [Wahbe93]. MiSFIT works as an assembler-level filter for several reasons. First, not writing a binary editing tool simplified the task tremendously, as there was no need to parse, disassemble, patch, and reassemble x86 binary code. Another motivation was that it conforms to the Unix tool-oriented approach for building systems. By not adding it to g++, MiSFIT has a degree of compiler independence. Although MiSFIT makes a (small) number of assumptions about the format of its input, it could easily be modified to work with output from other compilers, such as lcc or Microsoft C++.

MiSFIT takes the strategy of being platform specific and language neutral; the Java Virtual Machine is both platform neutral and language neutral. We found that any need we had for platform independence was outweighed by our need for high performance and the ability to write extensions in C++.

2 SFI Is Not Enough

MiSFIT is not a complete solution to the problem of protection from misbehaved extensions.

First, protection from errant writes and calls is not sufficient; the application or kernel must provide a safe interface to the extension, or a safe environment in which it can run. Protection against illegal stores is useless if the extension can call **bcopy**() with arbitrary arguments. Safe equivalents of many other commonly used routines, such as **read**(), **write**(), and **printf**() will also be needed.

Second, and more importantly, software fault isolation (or any other memory protection mechanism) is not a substitute for a resource management strategy. An extension should not be allowed to allocate memory, obtain a lock for a critical data structure, or even be given the freedom to run on the CPU, unless some mechanism is provided for the resource to be revoked if the extension fails to release it in a reasonable amount of time. In related work [Seltzer96], we explored wrapping each extension invocation in a transaction; if the extension aborted, or failed to complete promptly, our system could abort the transaction and nullify any changes made by the extension.

The third way in which MiSFIT is not a complete solution is that it, by itself, does not ensure that a given piece of binary code has been processed by MiSFIT. There are at least two methods for solving this problem. First, extension writers can distribute source code for their extensions, and the person installing the extension could compile and MiSFIT the code before installing it. This technique may be reasonable for installing operating system extensions, as is done now with loadable kernel modules in NetBSD and Linux.

The second method is more end-user-friendly, but is logistically more complex. Code processed by MiSFIT would be given a cryptographic digital signature, either by the tool itself or by a signing authority. This signature would then be checked at load time. In order to support this scheme it would be necessary to find a trustworthy authority willing to MiSFIT and sign code, or somehow safely hide the apparatus for generating the signature within MiSFIT itself.

Although there are pieces missing from MiSFIT to make it a complete environment for building extensible systems, they are both technically tractable and application specific. For our project (the VINO extensible operating system [Seltzer94]), we have developed a protected runtime environment, resource management infrastructure, and code signature scheme[2] for use with MiSFIT. Other applications of MiSFIT would necessarily have a different safe runtime environment and resource management infrastructure.

The remainder of this paper focuses on related work, the architecture of MiSFIT, and its runtime support. Section 3 contains a discussion of related work in extension technology. Section 4 discusses the design and implementation of MiSFIT, and Section 5 covers the related runtime support. Section 6 includes the overhead of MiSFIT on benchmark programs. Section 7 discusses what has been left out of MiSFIT, and the paper conclude in Section 8.

3 Related Work

The term Software Fault Isolation was introduced by Wahbe et al. [Wahbe93]. They proposed a type of software fault isolation, *sandboxing*, which has low overhead on a processor with a large number of registers.

2. Our code signature implementation uses the RSAREF library [RSA], which is export controlled.

Their tool was originally targeted for the MIPS and Alpha processors. The initial results for this work show overheads of roughly five percent to ten percent.

A follow-on to that work is the Omniware Portable Code system. The Omniware compiler generates portable code for an abstract virtual machine (OmniVM) which is translated to native fault-isolated code at runtime [Adl96]. Along with the source language independence provided by software fault isolation techniques, the Omniware system also offers target-independent portable code.

Silver has developed a version of gcc which generates software fault isolated code for the DEC Alpha processor [Silver96]. Most of the modifications to gcc were made in the machine-independent portion of the compiler, although some changes were needed in the machine dependent portion of the code. The author reports that the implementation is dependent upon a large number of registers being available for use by the tool; a port to x86, which has a severely limited register set, appears to be difficult, if not impossible.

Several other researchers in the area of extensible operating systems have developed one-off software fault isolation tools, including Banerji [Banerji96], Engler [Engler95], and Mazieres [Mazieres96]. Unfortunately these tools suffer from working on less widely used platforms, working only with domain-specific languages, or not being publicly available.

Some extensible systems designers have followed a different route, proposing that extensions be written in a safe language (e.g., the SPIN operating system [Bershad95], which uses Modula-3 [Nelson91], and Netscape Navigator, which uses Java [Gosling96]). Safe languages can perform as well or better than software-fault-isolated unsafe languages, but have the two disadvantages that there is no possibility of reusing existing C or C++ code, and that programmers need to develop extensions in the safe language, and not the more familiar and common unsafe languages. The performance overhead of Modula-3 relative to compiled C or C++ appears to be negligible, but Java (which is most often interpreted by a virtual machine) is 20 to 50 times slower than equivalent compiled C code [Small96].

The Netscape Navigator world-wide web browser is an interesting example of an extensible system. The current release (3.0) supports two types of extensions: those written in Java (a safe language) and JavaScript (an interpreted scripting language). In order for Netscape Navigator to support extensions written in Java on all platforms, a complete implementation of the Java interpreter and runtime environment must be developed on each platform. It is arguably less work to construct a simple software fault isolation tool for a hardware architecture than to develop or port an interpreter and runtime environment.

Although in previous work we have measured interpreted Java as running ten to seventy times slower than compiled C, several companies plan to release "just-in-time" native code compilers for Java[3]. These compilers would convert Java bytecode into native code as it is loaded (or first run). The overhead of running "just-in-time" compiled code has been measured at two to ten times that of regular compiled code [Hölze94], which would give Java roughly the same performance as software fault isolated code.

Microsoft offers the ActiveX extension mechanism, which provides no technical guarantee of safety, but instead supplies only a method for verifying the identity of the provider of the code through the use of digital signatures. Software fault isolation can work in concert with digital signatures, to guarantee both the identity of the provider and the safety of the code.

The design of the VINO extensible operating system, which is the primary testbed for MiSFIT, is described in more detail in other work [Seltzer94, Seltzer96].

4 MiSFIT Design and Implementation

Software fault isolation can be used to protect against illegal jumps, stores, and loads. Protecting against illegal stores and jumps is necessary for correctness, but protection from illegal reads is usually a security issue, not a correctness issue. (If an extension can read outside its memory bounds, it may be able to find data it should not be allowed to see, but if an extension can write or jump to an arbitrary location in memory, the stability and correctness of the host program can be compromised.[4])

MiSFIT can be used to fault isolate indirect loads, stores, and calls. It acts as a filter, sitting between the compiler and the assembler. MiSFIT scans the output of the compiler and builds an in-memory representation for the module. It then processes each instruction of the module in turn. If any implicitly unsafe instruction (e.g., **halt**) appears, the module is rejected. The arguments for each store, call, and (optionally) load instruction are examined. (Constants and general-purpose registers are implicitly safe.) Once the module has been processed, simple peephole optimization is performed (to remove

3. Symantec has shipped a just-in-time compiler, and Sun has announced plans to do so.

4. This is not necessarily the case inside the operating system kernel; on some hardware, such as the x86, device registers are mapped into memory and reset themselves after being read.

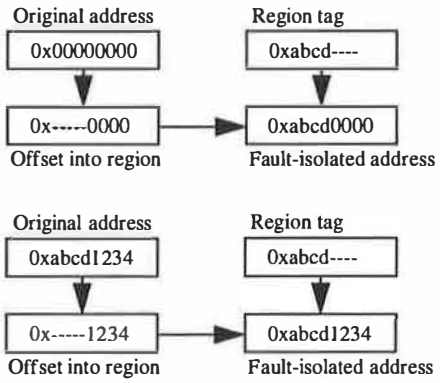


Figure 1: Example Transformations. In this example, the region tag is the top sixteen bits of the address and has the value 0xabcd. In the first example, the original address is invalid, so the fault-isolated address is different. In the second example, the original address is within the region, so the fault-isolated address is the same as the original address.

any redundancies introduced by the SFI transformation) and a new copy of the module is written out.

4.1 Indirect Loads and Stores

Loads and stores that use an indirect address that is computed at run-time are potentially unsafe. MiSFIT inserts code to *sandbox* [Wahbe93] arguments of these instructions to force the indirect address fall within a legal range.

Each user extension is assigned a contiguous region of memory into which it can write, and a region from which it can read. (These regions would normally at least overlap, if not be the same, but it is not necessary.)

MiSFIT requires that the size of each memory region be a power of two; because of this, the high bits of each address in the memory region (the *region tag*) will be the same. To sandbox a memory reference, MiSFIT simply sets the high bits of the reference to the region tag of its associated memory region. Any load or store that would have accessed memory outside its region is thus forced to fall somewhere inside the extension's memory region. Note that if the fault isolated target address was already in the extension's memory region, it does not change. The fault isolated address differs from the original target address only if the original target address was outside the extension's memory region (and therefore illegal). Examples of this transformation are given in Figure 1.

There is one more detail: in order to preclude the code from (unsafely) modifying itself, the writable

```
    movl eax,0(edx)          ; do the store
is transformed into:
    andl $0xffff,edx         ; clear old region tag
    orl destmask,edx         ; set our region tag
    movl eax,0(edx)          ; do the store

    movl eax,12(ebx,ecx)     ; do the store
is transformed into:
    pushl edx                ; obtain scratch register
    leal 12(ebx,ecx),edx     ; load target address
    andl $0xffff,edx         ; clear old region tag
    orl destmask,edx         ; set our region tag
    movl eax,0(edx)          ; do the store
    popl edx                 ; restore scratch register
```

Figure 2: Sandboxing transformations for a store instruction. In the first case the target is a simple indirection through a register; in the second case it is a complex indirection, so a scratch register is first made available and the target is loaded into the scratch register before sandboxing. In this example, the size of the assigned memory region is 64KB (the argument to the andl is 0xffff). Note that all of the added instructions take one cycle on the Pentium (assuming that the stack targets of the push and pop are in the first level cache). Note: the general format of x86 assembler instructions is *instr src, dest.*

region should be chosen so that it does not overlap the code space assigned to the extension.

MiSFIT modifies the loads and stores in the following way. First, it inserts code to load the target address into a register (if it is not already in a register). The high bits of the register are then cleared, and the region tag of the associated memory region is then OR'd into the register. The register is then used in place of the operand in the original instruction.

Depending on whether the target address was already in a register, this technique adds either two or five instructions[5]. If the original operand is an indirection through a single register (with no constant offset) only two instructions are needed, an AND to clear the high bits of the register and an OR to set the region tag. If the target address is not already in a register, MiSFIT inserts five instructions: MiSFIT obtains a scratch register (by pushing its current value on the stack), loads the effective target address into the scratch register, masks in the region tag as above, and restores the scratch register.

Examples of these transformations are shown in Figure 2. Note that in the second case it would be possible to save the scratch register push and pop if MiSFIT

5. Each instruction is executed in one cycle on the Pentium, assuming all memory references hit in the L1 cache.

were able to determine that there was a dead register[6] available that could be used as a scratch register. The MiSFIT performance impact is low enough that we have not yet been tempted to perform this optimization.

4.2 Virtual Function Calls

When a virtual function call takes place MiSFIT must verify that the target address is one that the extension is permitted to call. If the extension were allowed to indirectly call to any address, it not only might obtain access to an unsafe function, it also might jump into the middle of an instruction or into data space, which would open all sorts of security and safety holes.

MiSFIT restricts the extension by searching a table of valid function targets on each indirect call from an extension. The builder of the base system provides a file with the names of the functions that an extension may call; an auxiliary tool (provided with MiSFIT) determines the start address of each of these functions at link time, and places the addresses into a table that is linked into the base system.

Although there may be an arbitrarily large number of valid target addresses, the tool greatly limits search time by storing the valid addresses in a sparsely populated open addressed hash table [Cormen90]. An open addressed hash table is implemented as an array; the hash value of the key gives the index of the array to check. When the tool adds items to the hash table, if the key hashes to *n* and location *n* of the table is already in use, it check locations *n+1*, *n+2*, and so on, until it finds a free slot for the value. When searching for a key in the table, the search function hashes the key, yielding *n*, and then check location *n* of the table. If location *n* has a value (but not the key) it checks location *n+1*, *n+2*, and so on, until it either find the key (signifying success) or find an empty slot (signifying failure).

One subtle advantage of using an open addressed hash table is that if the search function does not find the key at location *n*, because the next location checked (at index *n+1)* is at an adjacent memory location, it is likely to be in the cache. So, even if it fails on the first probe of the table, the cost of subsequent probes is reduced.

By decreasing the density of the table, it is possible to reduce the number of probes needed nearly to unity (the theoretical minimum). With a table that has a 50% density (half the slots are empty) an average of fewer than 1.5 probes per indirect call are required. The overhead of each probe is roughly six to ten cycles (assuming everything hits in the L1 cache), adding, on average, approximately ten to fifteen cycles to each indirect function call.

Indirect calls are common in C++ code, as virtual functions are implemented as indirect calls. When protecting C++ code with MiSFIT the table of valid function targets can become quite large, but the per-invocation cost remains low, because the number of probes into the table is independent of the size of the table, depending only on its density, which is under MiSFIT's control.

4.3 Global Data, Virtual Function Tables

Because MiSFIT sandboxes global memory references, any data accessible to the extension must be placed in the memory region assigned to the extension. If there is global data that the extension should be able to access, the data should be placed in the memory region assigned to the extension. This applies not only to global program data, but other shared state, such as virtual function tables.

The restriction on global program data is a problem if multiple extensions are to be granted access to the same datum. A work-around is for the application to provide functions to access the data; each extension will be given permission to call these accessor functions, and use them instead of directly reading and writing the data.

This technique has an impact on performance that is difficult to quantify, as the cost is a function of the amount of data that is protected in this way, the frequency of access, and the type of interface the functions provide. In two of the three tests discussed in this paper this cost is not to quantified; in the third, the cost is built in to the overall model, but not factored and measured separately.

Virtual function tables are a different matter. If MiSFIT is configured to use read protection, virtual function tables need to be in a region of memory that is readable by the extension. The solution we have chosen for VINO is to store all virtual function tables in a contiguous region of memory (by making a one-line change to g++), and mapping that region into each extensions read-only region.

4.4 Block Instructions

The x86 instruction set includes memory-to-memory move and comparison instructions, **movs** and **cmps**, which take four or five clock cycles on the Pentium. The same goal can be accomplished by four one cycle instructions (assuming a scratch register is available). However, the memory to memory instructions have the advantage that they can be used to construct *block* move and compare sequences. The x86 **rep** instruction can be used as a prefix to the memory-to-memory instructions;

6. A dead register is one that will not be read again before it is written.

the **rep** prefix instructs the processor to repeat the memory-to-memory instruction for *count* times, where *count* is the value in the %ecx register. The block move instruction sequence has a lower per-move overhead than a sequence or loop of individual memory-to-memory move instructions, and can be generated by compilers to perform structure copies and in-line expansions of common C library functions such as **strcmp()** and **bcopy()**.

MiSFIT transforms the base addresses and repeat count of arguments to the block instruction, sandboxing the compound instruction as a whole. Although this adds a high fixed overhead to the block instruction (roughly 26 cycles), there is no per-element cost. The alternative, transforming the block instruction into a loop and sandboxing the instructions in the loop, has a high per-element overhead; the break-even point for the two techniques is at three or four iterations. Block instructions are typically used for copying or moving more than four elements, so the fixed overhead imposed by MiSFIT's technique is preferable.

4.5 Saved Registers and Return Addresses

Protecting the contents of the stack is also problematic. The stack is used not only for local variables (which must be accessible to the user extension) but also saved registers and the function return address (which should not be accessible to the user extension). If the user extension could write to arbitrary locations on the stack, the return address of the function could be overwritten and set to an arbitrary value, circumventing the call protection offered by MiSFIT.

A second problem is that the process stack is normally not in the same region of memory as the heap and global data; MiSFIT's technique depends on all valid memory references falling within a single region of memory. In a multi-threaded environment (either a multi-threaded operating system kernel or multi-threaded end-user application) each thread of control is assigned its own stack. In environments where the extension can be run as a separate thread of control, MiSFIT can co-locate the stack assigned to the thread (i.e. assigned to the extension) with the memory region assigned to the extension. Then all valid memory references made by the extension will fall within a single region.

In environments where there is a single thread of control, MiSFIT can provide the same type of protection by providing each extension with its own stack, located in its memory region. When the extension is invoked, the application switches to the stack associated with the extension. When the extension returns to the application, the process switches back to the original stack.

To solve the problem of an extension overwriting a return address on the stack, MiSFIT replaces each **call** instruction within the extension with a call to a support routine that saves the return address in a separate stack outside the extension's memory region and then jumps to the called function. MiSFIT then replaces each **ret** instruction with a jump to a second support routine that loads the saved return address and jumps to it. In this way, even if the extension misbehaves and overwrites the return address, the system returns to the correct location. To ensure that register values are preserved across the invocation of the extension, MiSFIT stores the contents of all callee-saved registers on entry to the extension, and reloads these values when it returns.

4.6 Dynamic Linking

MiSFIT modifies the operands of load, store, and call instructions that are computed at runtime. It does not modify operands that are labels, assuming that references to addresses within the module (i.e. local jumps, and loads and stores of module-level variables) are implicitly safe (generated by the compiler), and references to addresses outside the module will be checked by the dynamic linker when they are resolved. This implies that the dynamic linker is responsible for keeping track of which symbols may be linked to by an extension. Under some circumstances it may be the case that not all extensions will be given access to the same set of entrypoints. If this is the case, the dynamic linker is responsible for determining to which entrypoints a given extension should be given access.

Relinquishing responsibility for protecting external symbols has a limitation. The assembler does not mark external symbols as being for read or write use; a single external reference is generated for all reads and writes. If there is no read protection, but there is write protection, there is no way for the linker to discern which references are source (read) references and which are destination (write) references – in other words, which should be allowed, and which should be disallowed.

To solve this problem, MiSFIT generates a table of addresses of instructions that write operands that are labels. The dynamic linker can use the information in this table, in addition with the external reference table, to differentiate between read references and write references at link time.

4.7 An Alternative to Sandboxing on the x86

On the x86, an alternative to sandboxing exists. The **bound** instruction checks that a value falls within a specified range; if it does not, a trap occurs. If this trap can be caught, the ill-behaved extension can be stopped before it does any damage. It appears that the **bound** instruction was designed to be used for array bounds

checking. since it performs a signed (rather than unsigned) comparison. This does not preclude using it for SFI. MiSFIT might arrange for all parts of the region of memory assigned to an extension have the same sign (i.e. not cross the border between location 0x7fffffff and location 0x80000000), so the signed nature of the comparison would not be a problem. The bound instruction takes more cycles than the instructions needed to set the high bits of a register (eight vs. two); however, instead of neutering an illegal load or store the bound instruction would trap an illegal memory access.

This paper includes results of running tests comparing the performance of MiSFIT using the **bound** instruction and the sandboxing technique described in Section 4.1. The results show that the sandboxing method has superior performance, which is not surprising, considering the cost of the **bound** instruction in comparison with the cost of sandboxing.

5 Runtime Support

MiSFIT includes runtime support for linking extension code as new virtual functions to existing objects, setting up the state of an extension, and managing free store for the extension.

5.1 Virtual Function Table Manipulation

In the MiSFIT model, an extension is used to modify the behavior of a single object, by replacing a virtual function of that object. MiSFIT accomplishes this by making a copy of the virtual function table for that object and writing a new value into the slot corresponding to the replaced function.

The process by which an extension is called is somewhat baroque. MiSFIT can not just replace the address of the old function with the address of the newly loaded function in the virtual function table. As outlined above, when an extension is called its sandbox needs to be configured.

5.2 Calling The Extension

When an extension is installed, a small assembler stub function (similar to a closure) is created. This stub is responsible for configuring the sandbox and calling the extension. The stub is specific to the particular extension, because it includes the addresses of the extension's sandbox regions, as well as the address of the extension function itself.

The stub sets up the sandbox for the extension. It first saves callee-saved registers (as MiSFIT does not trust the extension to do so). The stub sets up the global variables that hold the region tags for the read and write (source and destination) regions assigned to the extension, and copies any arguments passed to the extension onto the extension's stack. It switches to the extension's stack, and jumps to the extension.

When the extension completes, it jumps to the returns stub (remember that the extension's **ret** instruction was replaced by this jump, as described in Section 4.5), which switches to the regular stack, loads the saved registers, and returns to the base system.

The runtime support code also includes the function that implements safe indirect calls (as described in Section 4.2). MiSFIT replaces indirect calls with code that loads the target function address and calls the hash table lookup function. If the function address is not found in the hash table by the lookup function, the function calls an **abort** function, which is responsible for cleaning up after the extension.

5.3 Extension Free Store Management

As the code running in an extension cannot reach outside its bounds, if it were to allocate storage using **new** it would not be able to read from nor write to that storage. MiSFIT provides a small heap in the data area assigned to the extension, and simple implementations of the built-in **new** and **delete** functions. When MiSFIT is processing an extension, it replaces any calls to the built-in **new** and **delete** functions with calls to the MiSFIT versions.

6 MiSFIT Overhead

This section compares the performance of unprotected code (written in C or C++) with the MiSFIT-protected versions. Times are reported as a percentage of the unprotected versions. Performance numbers for both write-call (where store and call instructions are protected) and read-write-call (where load, store, and call instructions are protected) tests are included. As pointed out above, read protection is typically a requirement for security, not for correctness.

6.1 Operating System Extensions

In previous work [Small96], we examined the suitability of various extension technologies for constructing operating system extensions. Three tests were developed and used, with each test representing a class of possible OS extensions. Following is a short description of each test; for more detail, the reader is directed to the earlier paper.

- *hotlist:* choose which page to evict from a linked list of page descriptors.
- *lld:* simulate the operation of a logical disk layer [DeJon93].
- *md5:* compute the MD5 checksum [RFC1321] of 1MB of data.

The tests were run on a 120MHz Pentium with 64MB of EDO memory, running BSD/OS 2.1. Each test

and its data fit into main memory. Times are reported relative to the unprotected version of the code. The results are found in Table 1.

The write-call overhead for these tests is low, at most 10%. The overhead for read-write-call protection can be much higher, over 200%.

In our earlier work we computed a break-even point for each operating system extension. If the cost of using the extension is below the break-even point, the extension will improve overall system performance; if it above this point, it will degrade system performance. The three write-call protected tests fall below the break-even point, as do the read-write-call versions of *lld* and *md5*, but the read-write-call version of *hotlist* does not.

Test	MiSFIT Write-Call Protected (MiSFIT/ unprotected)	MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected)
hotlist	1.00	3.2
lld	1.07	1.4
md5	1.09	1.7

Table 1: Relative overhead of MiSFIT-protected code to unprotected code on operating system extension benchmarks. The cost of isolating writes and indirect writes is low, under 10%, but the cost of protecting reads as well can be prohibitively high.

The performance of the write-call protected *hotlist* is equivalent to the unprotected version. This is because there are very few protected write instructions executed during the test. Because the kernel of the test repeatedly scans a linked list of page descriptors, the number of read instructions executed is very high. This bias is reflected in the performance of the read-write-call protected version of this test, where the overhead is more than 200%.

The *lld* test has a noticeable but small write-call overhead of 7%; read protection adds another 33%. This test is not as read-intensive as *hotlist*, so the added overhead of read protection is much lower. The *md5* test has similar performance characteristics, with a sub-10% write-call overhead, and an additional 60% overhead for read protection.

6.2 SPECInt92

This experiment shows the results of several SPECInt92 benchmarks processed by MiSFIT, using write-call and read-write-call protection. The performance of the MiSFIT-protected code relative to native code is reported in Table 2.

Although it is unlikely that anyone would want to load a SPEC benchmark into a web browser or database server, these results give a feeling for the overhead imposed by MiSFIT on "typical" code. (To better estimate the overhead imposed by MiSFIT, the tables only include time spent at user level.)

Test	MiSFIT Write-Call Protected (MiSFIT/ unprotected)	MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected)
compress	1.09	1.26
espresso	1.15	1.76
eqntott	1.02	1.68
li	1.17	1.61

Table 2: Overhead of protection on SPECInt benchmarks for MiSFIT, relative to unprotected code. MiSFIT times are the mean of ten runs. Standard deviations were less than 1%, except for *compress*, where it was 2.6%.

The write-call MiSFIT overhead for the SPEC92Int code is comparable to that of MiSFIT on the operating system extension benchmarks, ranging from a factor of 1.02 to a factor of 1.17. As is seen above, the overhead of read-write-call protection is higher than the overhead for write-call protection, on the order of 1.26 to 1.76. This overhead is large, but still substantially less than that of an interpreted language.

For memory-intensive applications, such as data copies, a higher overhead should be expected. The overhead seen is, of course, a function of the ratio of protected instructions to unprotected instructions.

6.3 VINO Kernel Extensions

MiSFIT is used to protect the VINO operating system kernel from misbehaved end-user extensions. We measured the performance overhead of MiSFIT on four kernel extensions [Seltzer96], and include these results here. For these tests we used MiSFIT for read-write-call protection, and the overhead shown is in line with the overhead seen above.

The *Read-ahead* extension specifies which disk block to read next, by returning a value found in its memory region. This code performs little computation, so the overhead imposed by protecting its loads and stores dominate its performance.

The *Page Eviction* extension is similar to the *hotlist* extension described in Section 6.1, but instead of searching a linked list it searches an array. Because there is less pointer chasing, the overhead imposed by MiSFIT is lower.

Each time the *Scheduling* extension is called it searches a list of 64 process IDs. Because the code that traverses the list is trusted code (is part of the base sys-

tem, outside the extension itself, unlike in the case of *hotlist*), the overhead of using MiSFIT is much lower[7].

The fourth extension, which performs a simple encryption of a data stream, is data intensive. It copies 8KB of data from an input buffer to an output buffer, applying a trivial (XOR-style) encryption to the data.

This extension was designed to be a worst-case test for MiSFIT, with little computation done between each data load and store. The MiSFIT version of the code takes slightly more than twice as long as the unprotected code. It is theoretically possible for MiSFIT-protected code of this form to take as much as six times as long as protected code (remember MiSFIT can add five instructions for each load and store), but it is difficult, if not impossible, to construct a real-world example where every instruction is a load or store. One possible case is when data is being copied directly from one buffer to another (as is done in this example), but the overhead seen here is 100%, not 500%. In the case of a straight data copy (using the x86 **rep; movs** instruction pair), MiSFIT uses a different technique for fault isolation which has lower overhead (see Section 4.4).

Test	MiSFIT Read-Write-Call Protected (MiSFIT/unprotected)
Read-ahead	2.5
Page Eviction	1.2
Scheduling	1.1
Encryption	2.1

Table 3: VINO Kernel Extensions: MiSFIT was used to apply read-write-call protection, which causes overhead in line with the results seen above. Each test was run between 300 and 3000 times; the standard deviation of each result was less than 2.5%.

6.4 Performance Summary

With read-write-call protection MiSFIT protected code can take from 1.4 to 3.2 times as long as unprotected code. Although this overhead may seem large, it should be compared to the overhead of an interpreted safe language, such as current Java implementations (which are 20 to 50 times slower than compiled C code), or the disadvantage of writing extensions in an unfamiliar, but safe, compiled language, such as Modula-3.

7. Calling trusted code outside the extension is analogous to a Java application calling native methods, which are implemented in compiled C or C++.

7 What Is Missing

As shown in Section 2, SFI is not a complete solution. The MiSFIT package does not include a safe runtime support library, which would be specific to the base system. This support library would be responsible for ensuring that extensions do not violate their resource limitations.

Extensions do not have access to the global heap; a version of **malloc** (or **new**) is needed that allocates memory from a pool inside the extension's writable memory region.

MiSFIT does not include a dynamic linker. Depending on its application, a dynamic linker may already be part of the system (e.g., NetBSD). The dynamic linker, or some code-signing tool, would be responsible for verifying that the loaded code had been processed by MiSFIT.

One restriction that is not currently addressed, but should be, is the difficulty of passing arguments and returns by reference. When calling an extension, the calling stub pushes arguments onto the extension's stack, but these arguments are currently restricted to immediate values. If the base system wants to pass an argument by reference (via a pointer) there is currently no way to do so. Additionally, there is no way for an extension to pass back data other than as the return value of the function or by storing the results in its writable memory region for later retrieval by the base system.

The solution to this limitation is the application of standard techniques for marshalling and unmarshalling arguments for remote procedure calls. By specifying the number and types of parameters to the extensions with an interface definition language, extension-specific stub functions could be generated that would copy arguments into the extension's address space when it is called, and copy results back to the base system when it returns.

8 Conclusions

The overhead imposed by MiSFIT when it is used for write and call protection is small. It allows applications and kernels to be protected from end-user extensions written in otherwise unsafe languages. Unlike other tools, it is freely available. As part of an end-to-end solution to the problem of constructing an extensible system, MiSFIT can provide safety at low cost.

9 Availability

MiSFIT is covered by a BSD-style license, and is available for public use without fee. Contact the author (chris@eecs.harvard.edu) to obtain a copy of the code.

10 Acknowledgments and Apology

The members of the VINO Operating System project, Prof. Margo Seltzer, Keith Smith, Yasuhiro Endo, and David Holland, are implicitly included and thanked wherever first-person plural is used in this paper. Keith Smith's eagle-eyed proofreading of the final draft was greatly appreciated by the author.

To circumvent use of both the awkward-sounding first-person singular and the royal "we," work done by the author has been ascribed either to the paper itself or directly to MiSFIT. Only where completely unavoidable was passive voice used. The author apologizes, profusely, to the linguistically offended.

Bibliography

[Adl96] Adl-Tabatabai, A., Langdale, G., Lucco, S., Wahbe, R., "Efficient and Language-Independent Mobile Programs," *PLDI '96*, Philadelphia, PA, 127-136, May 1996.

[Banerji96] Banerji, A., Panteleenko, V., Wyant, G., Cohn, D., "Quantitative Analysis of Protection Options," University of Dame Technical Report TR-96-20, 1996.

[Bloor96] Bloor, R., "The Capabilities of Illustra and its Integration with Informix DSA," http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/contents.htm

[Bershad95] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th SOSP*, Copper Mountain, CO, 267–284, December 1995.

[Cormen90] Cormen, T., Leiserson, C., Rivest, R., *Introduction to Algorithms*, 232-241, The MIT Press, Cambridge MA, 1990.

[DeJonge93] de Jonge, W., Kaashoek, M. F., Hsieh, W., "The Logical Disk: A New Approach to Improving File Systems," *Proc. 14th SOSP*, Asheville, NC, 15–28, December 1993.

[Engler95] Engler, D., Kaashoek, M. F., O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th SOSP*, Copper Mountain, CO, 251–266, December 1995.

[Fall93] Fall, K., Pasquale, J., "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," *1993 Winter USENIX Conference,* San Diego, CA, 327–334, January 1993.

[Gosling96] Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley, Reading, MA, 1996.

[Hölze94] Hölze, U., Ungar, D., "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback", *PLDI '94,* Orlando, FL, June 1994.

[Mazieres96] Mazieres, D., personal communication.

[Nelson91] Nelson, G., ed., *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm," *Network Working Group RFC 1321,* April 1992.

[RSA] ftp://ftp.rsa.com/rsaref

[Seltzer94] Seltzer, M., Endo, Y., Small, C., Smith, K., "An Introduction to the VINO Architecture," Harvard University Computer Science Technical Report TR-34-94, 1994.

[Seltzer96] Seltzer, M., Endo, Y., Small, C., Smith. K., "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," *Proc. 2nd OSDI,* Seattle, WA, 213–228, October 1996.

[Silver96] Silver, S., "Implementation and Analysis of Software-Based Fault Isolation," Dartmouth College Technical Report PCS-TR96-287, 1996.

[Small96] Small, C., Seltzer, M., "A Comparison of OS Extension Technologies," *Proc. 1996 USENIX Technical Conference,* New Orleans, LA, 41-54, January 1996.

[Wahbe93] Wahbe, R., Lucco, S., Anderson, T., Graham, S., "Efficient Software-Based Fault Isolation," *Proc. 14th SOSP,* Asheville, NC, 203–216, December 1993.

Krakatoa: Decompilation in Java
(Does Bytecode Reveal Source?)

Todd A. Proebsting Scott A. Watterson
The University of Arizona *

Abstract

This paper presents our technique for automatically decompiling Java bytecode into Java source. Our technique reconstructs source-level expressions from bytecode, and reconstructs readable, high-level control statements from primitive **goto**-like branches. Fewer than a dozen simple code-rewriting rules reconstruct the high-level statements.

1 Introduction

Decompilation transforms a low-level language into a high-level language. The Java Virtual Machine (JVM) specifies a low-level bytecode language for a stack-based machine [LY97]. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. Furthermore, the bytecode must be *verifiably well-behaved* in order to ensure safe execution. Decompilation systems can exploit this type information and well-behaved property to recover Java source code from the class file.

We present a technique for transforming low-level Java bytecode into legal Java source code. Our system, Krakatoa,[1] performs type inference to issue local variable declarations. The verifier does the same type of type inference, and the techniques are well known. Presently, we focus our research on two subproblems: recovering source-level expressions and synthesizing high-level control constructs from **goto**-like primitives.

Krakatoa uses a stack-simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations. We extend Ramshaw's **goto**-elimination algorithm to structure (and create source for) arbitrary reducible control flow graphs. This technique produces source code with loops and multi-level **break**'s. Subsequent techniques recover more intuitive constructs (e.g., **if** statements) via application of simple code rewrite rules.

Traditional decompilation systems use graph transformations to recover high-level control constructs. These systems require the author of the decompiler to anticipate all high-level control idioms. When faced with an unexpected language idiom, these systems either abort, or produce **goto**s (illegal in Java). Krakatoa represents a different approach. Krakatoa first produces legal Java source given legal Java bytecode with arbitrary reducible control flow, and *then* recovers intuitive high-level constructs from this source.

Figure 1 gives the five steps of decompilation performed by Krakatoa. First, the *expression builder* reads bytecode, recovers expressions and type information, and produces a control flow graph (CFG). Next, the *sequencer* orders the CFG nodes for Ramshaw's **goto**-elimination technique. Ramshaw's algorithm produces a convoluted—yet legal—Java abstract syntax tree (AST). Our system then transforms this AST into a less convoluted AST using a set of simple rewrites. The final phase produces Java source by traversing the AST.

Address: Department of Computer Science, University of Arizona, Tucson, AZ 85721; Email: {todd, saw}@cs.arizona.edu.

[1] Krakatoa is a volcano located in the Sunda Strait between Java and Sumatra. Its 1983 eruption threw five cubic miles of debris into the air and was heard 2200 miles away in Australia.

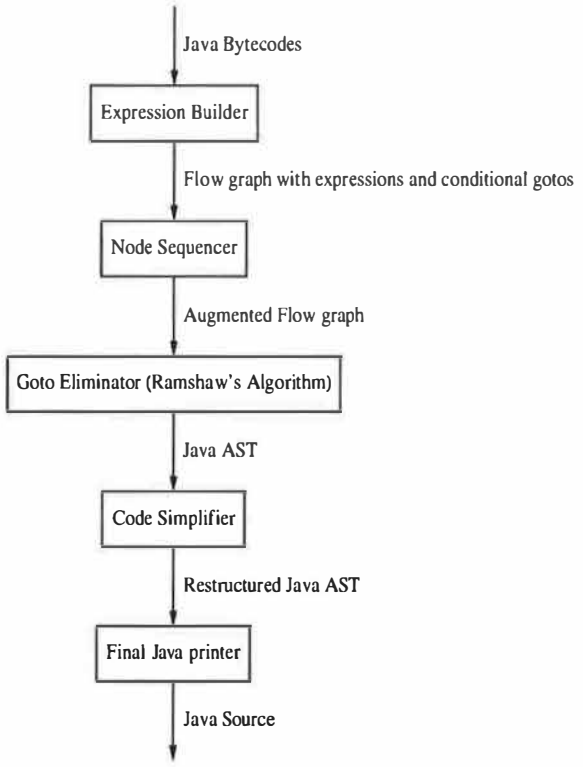


Figure 1: Java Bytecode Decompilation System

```
class foo {
   int sam;

   int bar(int a, int b) {
      if (sam > a) {
         b = a*2;
      }
      return b;
   }
}

Compiled from foo.java
class foo extends java.lang.Object {
    int sam;
    int bar(int,int);

Method int bar(int,int)
   0 aload_0
   1 getfield #3 <Field foo.sam I>
   4 iload_1
   5 if_icmple 12
   8 iload_1
   9 iconst_2
  10 imul
  11 istore_2
  12 iload_2
  13 ireturn
}
```

Figure 2: Simple Method and Bytecode Disassembly (via `javap -c`).

2 Expression Recovery

Java bytecodes bear a very close correspondence to Java source. As a result, recovering expressions from Java bytecode is often simple—much simpler than recovering expressions from machine language. Java class files include information that makes recovering high-level operations like field references easy. The fact that the bytecode must be well-behaved (i.e., verifiable) also simplifies analysis. Figure 2 gives a sample program and its abbreviated disassembly. Note the level of type information in the disassembly produced by Sun's **javap** utility.

Symbolic execution of the bytecode creates the corresponding Java source expressions. It also creates conditional and unconditional **goto**'s, which will be removed by subsequent decompilation steps. Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed. For instance, `iload_1`, which loads the value of the first local variable—with type `int`—could be represented on the stack as "`i1`". Similarly, if `i1` and `2` were the top two elements of the symbolic stack, and the next bytecode were `iadd` (integer addition), those elements would be popped off the stack and replaced with "`(i1+2)`". The symbolic execution of some expressions, like assignment, requires *emitting* Java source.

Our algorithm recovers expressions one basic block at a time. Some basic blocks (such as those produced by the conditional expression operation, *A?B:C*) do not begin with empty stacks, so some information is required to propogate from predecessors. Also, basic blocks that begin exception-handling blocks—which are easily identified—begin with the raised exception on the stack.

Figure 3 provides the step-by-step decompilation of the bytecode in Figure 2. The initial `aload_0`

instruction pushes a Java reference onto the stack. In virtual functions, the "0'th" local variable, `a0`, always refers to `this`. The `getfield` instruction references a *named* field, "`sam`", of the current top of stack. Therefore, the "`this`" is popped and replaced with "`this.sam`". `iload_1` pushes "`i1`" onto the stack. The `ifcmple` compares the top two stack elements and branches to the appropriate instruction if the lower is less than or equal to the top element. Symbolically executing the `ifcmple` requires popping the top two elements and emitting the appropriate conditional branch. Translating the remaining instructions is similar.

Most of the bytecode instructions are equally simple to symbolically execute. Unfortunately, a few require more information. Some of the stack manipulation routines (e.g., `pop2`, `dup2`, etc.) depend on byte offsets from the stack top. For instance, `pop2` removes the top 8 bytes from the stack, whether those 8 bytes represent *one* 8-byte double value, or *two* 4-byte scalar values. To correctly simulate these instructions the symbolic execution keeps track of the size (and type) of each stack element.

3 Instruction Ordering

After recovering expressions, conditional and unconditional **goto**'s (along with implicit *fall through* behavior) determine control flow. Java, however, has no **goto** statement, so its control flow must be expressed with structured statements.

Ramshaw presented an algorithm for eliminating **goto**'s from Pascal programs while preserving the program's structure [Ram88]. This algorithm replaces each **goto** with a multilevel **break** to a surrounding loop. The algorithm determines the appropropriate locations for these surrounding loops. We trivially extended his algorithm to use multilevel **continue**'s.

Ramshaw's (extended) algorithm replaces each forward **goto** with a **break** and each backward **goto** with a **continue**. His algorithm inserts a loop that ends just before the target of each **break** statement. Likewise, it inserts a loop that starts just before the target of each **continue**. These loops ensure that each control-transfer statement jumps to the correct instruction. Each newly-inserted loop must also end with a **break** statement, so that control will *fall* out of the loop. Figure 4 shows an example of this technique. Additional loops and **break/continue**'s create a structured program with exactly the same control flow as the **goto**-only program.

Ramshaw's algorithm requires two inputs: the control flow graph, and an instruction ordering. His algorithm encodes this order into the flow graph using *augmenting edges*, such that every instruction has an augmenting edge to the next instruction in sequence. These augmenting edges occur between every pair of physically adjacent instructions even if actual control flow between them is impossible. He proves that if this augmented graph is reducible, then a *structurally equivalent* [PKT73] program can be created without **goto**'s. However, Ramshaw provides no algorithm for finding a reducible augmented flow graph from a given reducible flow graph.

The control-flow graphs of Java programs are reducible. Therefore, the compiled bytecode will likely form a reducible control-flow graph. Unfortunately, simple optimizations like loop inversion create irreducible augmented flow graphs. The flow graph of the program in Figure 8 has this problem because the augmenting edge between the first two statements creates a "jump" into the body of the loop formed by the next seven statements.

To utilize Ramshaw's algorithm, we developed an algorithm that orders a reducible graph's instructions such that the resulting augmented graph is also reducible.

3.1 Augmenting the Flow Graph

Creating a reducible augmented flow graph requires that no augmenting edge enters a loop anywhere other than at its header. Preventing this is simple—when ordering the instructions, make the header first and contiguously order the loop's instructions. Because physical adjacency determines augmenting edges, contiguously ordering the instructions guarantees that the only augmenting edge entering the loop from the outside will be entering at the top, which will not affect reducibility if it is the loop's header.

A loop with no nested loops inside is easy to order—simply remove the back edges and topologically sort the remaining directed acyclic graph (DAG). Handling interior loops requires replacing them with a single *placeholder* node in the graph and separately ordering both the loop and the surrounding graph. After ordering both, re-insert the loop's nodes at its placeholder. Re-ordering instructions may change whether or not one instruction *falls through* to another as it did in the original

Bytecode		Symbolic Stack	Emitted Source
	aload_0	"this"	
	getfield #3 <Field foo.sam I>	"this.sam"	
	iload_1	"this.sam", "i1"	
	if_icmple 12		if (this.sam <= i1) goto L12
	iload_1	"i1"	
	iconst_2	"i1", "2"	
	imul	"(i1*2)"	
	istore_2		i2 = (i1*2)
12:	iload_2	"i2"	L12:
	ireturn		return i2

Figure 3: Symbolic Execution of Bytecode

*stmt*0
if *expr*1 **goto** L1;
if *expr*2 **goto** L2;
L1: *stmt*1
L2: *stmt*2

*stmt*0
L2: **for** (; ;) {
L1: **for** (; ;) {
if *expr*1 **break** L1;
if *expr*2 **break** L2;
break L1;
} // L1
*stmt*1
break L2;
} // L2
*stmt*2

Figure 4: Ramshaw's Goto Elimination: Before and After

ordering. Where implicit control flow has changed, the algorithm must add new branches to restore the original control flow. Whenever possible, the topological sort attempts to maintain the original fall-through behavior.

This algorithm produces a reducible augmented graph. Because all loops are ordered separately, and laid out contiguously, the only augmenting edge entering from outside enters at the top. The topological sort of the loop (minus its backedges) guarantees that this top node is the loop header and that no internal edges cause irreducibility. Outside edges into the loop header cannot make a loop irreducible. Therefore, the resulting augmented graph is reducible.

Loops are not the only blocks of instructions which must be ordered contiguously. Exception handling regions must form contiguous sections of instructions. Class files specify which instructions are in which regions. Our algorithm orders those regions contiguously by treating them like loops.

After applying this technique to create a total ordering of the nodes (the augmenting path), Krakatoa can apply Ramshaw's technique to eliminate **goto**'s.

4 Code Transformations

4.1 Program Points

After applying Ramshaw's algorithm for eliminating **goto**'s, Krakatoa has a complex, yet legal, Java AST (see Figure 9). Krakatoa then proceeds to recover more of the natural high-level constructs of the original program (e.g. **if-then-else**, etc.). Krakatoa uses a *program point* analysis to summarize a program's control-flow and to guide recovering high-level constructs. A program point is a syntactic location in a program. Every statement has a program point both before and after it. These program points have two properties: *reachability* and *equivalence class.*

A program point is unreachable if and only if it is preceded along all execution paths by an uncondi-

tional jump statement (i.e. **return**, **throw**, **break**, or **continue**). For instance, in Figure 5, program point 3, Φ_3, is unreachable, since it is preceded by a **return** statement. Φ_6 is reachable, however, since one of the branches in the preceding **if** statement does not end with a jump statement.

Two program points are equivalent (denoted as $\Phi_x \approx \Phi_y$) if and only if future computation of the program is the same from both points. For instance, the program point before a **break** statement is equivalent to the program point after the loop it exits (Φ_3 and Φ_8 in Figure 6). As an example, in Figure 6, Φ_1, Φ_2, Φ_4, Φ_5, Φ_6, and Φ_7 are equivalent, as are program points Φ_3 and Φ_8.

Both reachability and equivalence are simple to compute via standard control-flow analyses [ASU86].

4.2 AST Rewrite Rules

Krakatoa performs a series of AST rewriting transformations to recover as many of the "natural" program constructs as it can (e.g. **if-then-else**, etc.). Krakatoa applies these rewriting rules repeatedly until no changes occur. We have found that the few rules below are sufficient to retrieve high-level constructs of the Java language, including **if-then-else** statements, and short-circuit evaluation of expressions. Each rewriting rule reduces the size of the AST, thus ensuring termination.

Table 1 summarizes the rules, which we describe below in greater detail. Many of these rules generalize. Those that apply to **for**-loops often apply to other loops. Many rules have several symmetric cases. For example, the first rule in Table 1 removes an empty **else**-branch from an **if-then-else** statement—there is a symmetric rule for removing an empty **then**-branch by negating the predicate.

4.3 if-then-else Rewriting Rules

The first transformation shown in Table 1 changes an **if-then-else** statement into an **if-then** statement when the **else** branch is empty. This transformation is always legal.

The second transformation creates an **if-then-else** statement from an **if-then** statement by hoisting the subsequent statement list into the **else**-part. Our algorithm performs this transformation if and only if no reachable program point in *Stmtlist1* is equivalent to the program point before *Stmtlist2*. Essentially, this means that no statement in the **then**-branch (*Stmtlist1*) can reach *Stmtlist2* directly.

4.4 Loop Rewriting Rules

The third rule in Table 1 removes useless **continue** statements. If the program point after a **continue** statement is equivalent to the program point before the **continue** statement, then that **continue** can be removed.

The fourth rule creates a short-circuit test expression within a **for**-loop by eliminating an interior **if** statement. Doing so requires that the loop body begin with an **if-then-else** statement, and that the **then** branch of that statement consists of a single jump to a program point equivalent to breaking out of the loop.

The fifth transformation provides an example of transforming loops into **if** statements. A loop is equivalent to an **if** if it can never repeat itself, and if all simple **break** statements can be safely removed during the transformation. A loop never repeats if its last program point is unreachable. **break**'s may be removed if the immediately following (unreachable) program point is equivalent to the last program point in the loop (Φ_1 in Table 1). The transformation replaces the loop with an **if** statement, and deletes all of the **break** statements for that loop.

4.5 Short Circuit Evaluation Rewriting Rules

The sixth rule shown in Table 1 recovers a short-circuit **Or** conditional. Short-circuit **Or**'s exist when two adjacent conditionals guard the same statement list and failure of either will cause a branch to equivalent locations.

The last transformation in Table 1 recovers short-circuit **And** expressions. This transformation is applicable whenever a simple **if** statement represents the entire body of another.

5 Status

We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class files, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well. Figures 7–10 provide an example of the stages of decompilation.

Rule	Before	After	Conditions

Eliminate Else	**if** *expr* {
	Stmtlist
	} **else** { }
		if *expr* {
		Stmtlist
		}
			None

Create **if-then-else**	**if** *expr* {
	Stmtlist1
	}
	Φ_1 : *Stmtlist2*
		if *expr* {
		Stmtlist1
		} **else** {
		Stmtlist2
		}
			Stmtlist1 contains no reachable program points equivalent to Φ_1.

Delete Continues	**for** (*A* ;*B* ;*C*) {
	Stmtlist
	Φ_1 **continue** Φ_2
	}
		for (*A*;*B* ;*C*) {
		Stmtlist
		}
			$\Phi_1 \approx \Phi_2$

Move Conditionals	**for** (*A* ;*B* ;*C*) {
	if *expr* {
	Φ_1 **jump**
	} **else** {
	Stmtlist1
	}
	Stmtlist2
	}
	Φ_2
		for (*A* ;
		B **and not** *expr* ;
		C) {
		Stmtlist1
		Stmtlist2
		}
			$\Phi_1 \approx \Phi_2$, X is either a **break** or **continue**

Remove Loop	**for** (*stmt* ; *expr* ;) {
	Stmtlist
	Φ_1
	}
	Φ_2
		stmt
		if *expr* {
		Stmtlist'
		}
			Stmtlist contains no reachable program points equivalent to Φ_1. The program point after any **break** must be equivalent to Φ_1.

Create Short Circuit Or's	**if** *expr1* {
	Φ_1 *X*
	} **else** {
	if *expr2* {
	Φ_2 *Y*
	} **else** {
	Stmtlist
	}
	}
		if *expr1* **or** *expr2* {
		X
		}
		else {
		Stmtlist
		}
			X and *Y* are equivalent jumps. (I.e., $\Phi_1 \approx \Phi_2$.)

Create Short Circuit And's	**if** *expr1* {
	if *expr2* {
	Stmtlist
	}
	}
		if *expr1* **and** *expr2* {
		Stmtlist
		}
			Neither **if** stmt has an **else** branch

Table 1: Canonical Code Transformation Rules

```
Φ1
if ( a < b ) {
    Φ2
    return a;
    Φ3 // (unreachable)
}
else {
    Φ4
    a = b;
    Φ5
}
Φ6
```

Figure 5: Reachable Points

```
Φ1 // { Φ2, Φ4, Φ5, Φ6, Φ7 }
for ( ; ; ) {
    Φ2
    if ( a < b ) {
        Φ3 // { Φ8 }
        break;
        Φ4 // (unreachable)
    }
    else {
        Φ5
        continue;
        Φ6 // (unreachable)
    }
    Φ7 // (unreachable)
}
Φ8
```

Figure 6: Equivalent Points

Figure 7 shows the original source code of a sample program. Figure 8 shows the results of expression decompilation on the bytecode of this program. Figure 9 shows the results of applying Ramshaw's algorithm to the decompiled expression graph. Figure 10 shows the result of the grammar rewriting rules applied to the output of Ramshaw's algorithm. Obviously, using DeMorgan's laws would simplify the boolean expressions. Future versions of Krakatoa will do so.

For the JVM **dup** operators, which duplicate stack elements, Krakatoa simply creates a temporary variable to hold the duplicated value. This yields unnatural, but easily readable, decompilations. A more difficult problem is our failure to recover the conditional-expression operator, "**? :**". This operation presents two difficulties: it requires determining short-circuit operators during expression recovery, and it requires that expression recovery handle non-empty stacks at basic block boundaries. Fortunately, the short-circuit problem can be handled easily with four simple graph-writing rules given in [Cif93]. The non-empty stack problem is difficult because it requires combining expressions in our symbolic stack upon entering a basic block with multiple predecessors. Krakatoa again uses a temporary variable to hold the result of each branch of the conditional expression, and then assigns this temporary value to the conditional expression. We are currently investigating other solutions to this problem.

Appendix B contains additional examples of Krakatoa's output.

6 Countermeasures

Krakatoa is very effective at reproducing readable Java source from Java bytecode. This may be alarming to those who want to protect their source code from unwanted copying. Unfortunately, there are few countermeasures.

One could introduce irreducible control-flow through bogus conditional jumps to foil Ramshaw's algorithm. This, however, only stops the recreation of high-level constructs. Krakatoa could simply produce source code in a Java-like language extended with **goto**'s.

One could introduce bizarre stack behavior to foil expression recovery. This is difficult, however, because the behavior cannot be so bizarre as to yield unverifiable bytecode. It is possible, however, to create many bogus threads of control (i.e., threads that will never execute) that will confuse the expression recovery mechanism in basic blocks that are entered with non-empty stacks.

One code obfuscation technique that is modestly effective is to change the class file's symbol table to contain bizarre names for fields and methods. So long as cooperating classes agree on these names, the class files will link and execute correctly [vV96, Sri96].

Another suggested solution is to use dedicated

```
class foo {
  void foo(int x, int y) {
    while ((x + y < 10) && (x > 5)) {
      if ((y > x) || (y < 100)) {
        x = y;
      }
      else {
        x += 100;
      }
    }
  }
}
```

Figure 7: Original Source

```
class foo {
  void foo(int i1, int i2) {
          goto L4;
    L1:   if (i2 > i1) goto L2;
          if (i2 >= 100) goto L3;
    L2:   i1 = i2;
          goto L4;
    L3:   i1 += 100;
    L4:   if ((i1+i2)>=10) goto L5;
          if (i1 > 5) goto L1;
    L5:   return;
  } // foo
} // foo
```

Figure 8: After Expression Decompilation

```
class foo {
  void foo(int i1, int i2) {
lp3: for ( ; ; ) {
       if ((i1 + i2) >= 10) break lp3;
       if !((i1 > 5)) break lp3;
lp2 :  for ( ; ; ) {
lp1 :    for ( ; ; ) {
           if (i2 > i1) break lp1;
           if !((i2 >= 100)) break lp1;
           break lp2;
         } // lp1
         i1 = i2;
         continue lp3;
       } // lp2
       i1 += 100;
       continue lp3;
     } // lp3
     return;
   }
}
```

Figure 9: After Goto Elimination (Ramshaw's Algorithm)

```
class foo {
    void foo(int i1, int i2) {
lp3:   for ( ;!((i1+i2)>=10)&&((i1>5)); ) {
         if (i2 > i1) || !((i2 >= 100)) {
           i1 = i2;
         } // then
         else {
           i1 += 100;
         }
       } // lp3
       return;
    }
}
```

Figure 10: After AST Transformation (Final Decompilation Results)

hardware and encryption to protect class files [Wil97].

Many traditional countermeasures to reverse-engineering will not work for Java bytecode. It is impossible to mix code and data. It is impossible to jump to the middle of instructions. It is impossible to generate bytecode and then jump to it.

7 Related Work

Ramshaw presented a technique for eliminating **goto**'s in Pascal programs by replacing them with multilevel **break**'s and surrounding loops [Ram88]. He made no attempt to recover high-level control constructs. All high-level control structures were provided by the original Pascal.

Several decompilation systems have used a series of graph transformations to recover high-level constructs [Lic85, Cif93]. These systems encounter difficulties in the presence of nested loops, and other arbitrarily control flow. Multilevel **break**'s cause considerable problems. Exception handling introduces another difficulty to such systems, as the control flow graph can be entered in several places. Krakatoa easily creates multi-level **break**'s and **continue**'s, and is able to eliminate virtually all of the unnecessary ones via successive application of the rewrite rules.

"Mocha" (version 1 beta 1) [vV96] is a Java decompiler written by Hanpeter van Vliet. Mocha uses graph transformations to recover high-level constructs. Mocha often aborts when it confronts tangled—yet structured—control flow (including multi-level **break**'s and **continue**'s). The system does issue type declarations, and uses debugging information (when present) to recover local variable names.

Other graph transformation systems used node-splitting to transform an unstructured graph to a structured graph [WO78, PKT73, Wil77]. Peterson, Kasami, and Tokura present a proof that every flow graph can be transformed into an equivalent well-formed flow graph. Williams and Ossher use a similar technique, but they recognize five unstructured sub-graphs, and replace those with equivalent structured graphs. Node-splitting preserves the execution sequence of a program, but not the structure. We do not consider this reasonable for decompilation.

Baker presents a technique for producing programs from flow graphs [Bak77]. Baker generates summary control flow information to guide her graph transformations. Our goal is similar, since the output of the decompiler should be as readable as possible. Her technique structures old FORTRAN programs for readability. As a result, her technique may leave some **goto**'s in the resulting programs, which is not allowed in Java.

Other techniques for eliminating **goto**'s have been proposed [EH94, Amm92, AKPW83, AM75]. These techniques may change the structure of the program, and may add condition variables, or create subroutines.

8 Conclusion

In this paper, we present a technique for decompiling Java bytecode into Java source. Our decompiler, Krakatoa, produces syntactically legal Java source from legal, reducible Java bytecode. We focus on two subproblems of decompilation: recovery of expressions from Java's stack-based bytecode, and recovery of high-level control-flow constructs. We present our stack simulation method for recovering expressions. We present an extension of Ramshaw's **goto** elimination technique that can be applied to any reducible control-flow graph.

We also present a small, yet powerful, set of code rewriting rules for recovering the natural high-level control-flow constructs of the Java source language. These rewrite rules enable Krakatoa to successfully decompile many class files that graph transformation systems fail. If Krakatoa is presented with a high-level language idiom that it does not recognize, it may leave unnecessary **breaks** or **continues** in the code. It will still produce legal Java, however. If a system relies on a graph transformation system to produce high-level constructs, it will fail when presented with an unexpected construct.

Our techniques, combined with the abundant type information available in class files, make decompilation of Java bytecode quite effective.

9 Acknowledgment

Saumya Debray helped develop the instruction ordering algorithm.

References

[AKPW83] J.R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conver-

sion of control dependence to data dependence. pages 177–189, 1983.

[AM75] E. Ashcroft and Z. Manna. Translating programs schemas to while-schemas. *SIAM Journal of Computing*, 4(2):125–146, 1975.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(2):237–250, 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[Bak77] Brenda S. Baker. An algorithm for structuring flowgraphs. *Journal of the Association for Computing Machinery*, 24(1):98–120, January 1977.

[Cif93] Cristina Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informatica*, pages 267–276, Buenos Aires, Argentina, August 1993.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. pages 229–240. International Conference on Computer Languages, May 1994.

[Lic85] Ulrike Lichtblau. Decompilation of control structures by means of graph transformations. In C. F. M. Nivat Hartmut Ehrig and J. Thatcher, editors, *Mathematical foundations of software development: Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT 85): volume 1 - Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 284–297. Springer-Verlag, March 1985.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1997.

[PKT73] W.W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[Ram88] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the Association for Computing Machinery*, 35(4):893–920, October 1988.

[Sri96] KB Sriram. Hashjava. url: http://www.sbktech.org/hashjava.html, 1996.

[vV96] Hanpeter van Vliet. Mocha. current url: http://www.brouhaha.com/~eric/computers/mocha-b1.zip, 1996.

[Wil77] M.H. Williams. Generating structured flow diagrams: The nature of unstructuredness. *Computer Journal*, 20(1):45–50, 1977.

[Wil97] U. G. Wilhelm. Cryptographically protected objects, May 1997. A french version appeared in the Proceedings of RenPar'9, Lausanne, CH. `http://lsewww.epfl.ch/~wilhelm/CryPO.html`.

[WO78] M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Computer Journal*, 21(2):161–167, 1978.

A Additional Rewriting Rules

We anticipate using a few other tree rewriting rules that might improve readability of our code. The anticipated rules build more natural **for**-loops. Table 2 presents addition code transformation rules that could be applied by Krakatoa. We expect to add these rules as we re-implement Krakatoa in Java.

B Sample Decompiler Output

We've included a representative sampling of Krakatoa's output on a classfile that implements sets in Java. The original Java source is on the left and Krakatoa's output is on the right. Table 3 provides original source class definitions as well as the

Rule	Before	After	Conditions

Include Init	*I*
	for (; *expr* ; *update*) {
	Stmtlist
	}
		for (*I* ; *expr* ; *update*) {
		Stmtlist
		}
			I is a simple statement

Include Update	**for** (*init* ; *expr* ;) {
	Stmtlist
	U
	Φ_1 }
		for (*init* ; *expr* ; *U*) {
		Stmtlist
		}
			Stmtlist contains no reachable program points equivalent to Φ_1. *U* is a simple statement

Table 2: Additional Code Transformation Rules

Original Source	Output from Krakatoa

`import java.io.PrintStream;`
`import java.util.Vector;`
 	
`public class Set`
`  implements Cloneable {`
 	
`  // class variables`
`  static boolean echo_ops;`
 	
`  // instance variables`
`  protected Vector members;`
 	
`  // functions are defnied here....`
 	
`}`
	`import java.io.PrintStream;`
	`import java.util.Vector;`
	
	`public class Set`
	`  extends java.lang.Object`
	`  implements java.lang.Cloneable {`
	
	`  static boolean echo_ops;`
	
	`  protected java.util.Vector members;`
	
	`  // functions are defined here...`
	
	`}`

Table 3: Class definition output from Krakatoa

corresponding Krakatoa output. Table 4 provides original source of several small functions together with Krakatoa output for those functions. Table 5 shows a larger function in original source as well as Krakatoa output for that function.

Original Source	Output from Krakatoa

`public boolean isMember(Object o) {`
 	
`  return (members.contains(o));`
`} // isMember`
	`public boolean isMember(`
	`  java.lang.Object local1) {`
	`  return this.members.contains(local1);`
	`} // isMember`

`public void addMember(Object o) {`
 	
`  if (!(isMember(o))) {`
`  members.addElement(o);`
`  } // then`
 	
`} // addMember`
	`public void addMember(`
	`  java.lang.Object local1) {`
	
	`  if !( (this.isMember(local1) != 0)  ) {`
	`  this.members.addElement(local1)`
	`  } // then`
	`  return;`
	`} // addMember`

`public void removeMember(Object o) {`
 	
`  members.removeElement(o);`
 	
`} // removeMember`
	`public void removeMember(`
	`  java.lang.Object local1) {`
	
	`  this.members.removeElement(local1)`
	`  return;`
	`} // removeMember`

`public int size() {`
`  return members.size();`
`}  // size`
	`public int size() {`
	`  return this.members.size();`
	`} // size`

`boolean equals(Set s) {`
`  Set d1, d2;`
 	
`  d1 = difference(s);`
`  d2 = s.difference(this);`
 	
`  return ((d1.size() == 0) &&`
`  (d2.size() == 0));`
 	
`}`
	`boolean equals(Set local1) {`
	`  Set local2;`
	`  Set local3;`
	
	`  local2 = this.difference(local1);`
	`  local3 = local1.difference(this);`
	`  if !(((local2.size() != 0)  ||`
	`  !((local3.size() == 0)))) {`
	`  return 1;`
	`  } // then`
	`  else {`
	`  return 0;`
	`  } // if`
	`} // equals`

Table 4: Member Functions: Original Source and Krakatoa output

Original Source	Output from Krakatoa

`// This returns a NEW set, with all of`
`// the elements from this set and`
`// Set s.`
`public Set union(Set s) {`
`  Set out;`
`  int size;`
`  int i;`
`  Object obj;`
 	
`  if (echo_ops) {`
`  System.out.println("unioning");`
`  }`
 	
`  out = new Set();`
 	
`  out.members = (Vector) members.clone();`
 	
`  size = s.size();`
 	
`  for (i = 0; i < size; i++) {`
`  obj = s.members.elementAt(i);`
`  if (!(out.isMember(obj))) {`
`  out.addMember(obj);`
`  } // then`
 	
`  } // for`
 	
`  return out;`
`} // union`
	`public Set union(Set local1) {`
	`  Set local2;`
	`  int local3;`
	`  int local4;`
	`  java.lang.Object local5;`
	
	`  if !( (Set.echo_ops == 0)  ) {`
	`  java.lang.System.out.println("unioning");`
	`  } // then`
	
	`  local2 = new Set();`
	
	`  local2.members = ((java.util.Vector)`
	`  this.members.clone());`
	
	`  local3 = local1.size();`
	`  local4 = 0;`
	`  loop3 :`
	`  for ( ; !(!(local4 < local3)) ; ) {`
	`  local5 =`
	`  local1.members.elementAt(local4);`
	`  if !((local2.isMember(local5) != 0)) {`
	`  local2.addMember(local5)`
	`  } // then`
	`  local4 += 1;`
	`  } // loop3`
	
	`  return local2;`
	`} // union`

Table 5: Member functions: Original Source and Krakatoa output

Resource Access Control for an Internet User Agent

Nataraj Nagaratnam, *Dept. of ECE, Syracuse University*
Steven B. Byrne, *JavaSoft, Inc., Sun Microsystems*
email: { *nataraj@cat.syr.edu, sbb@eng.sun.com*}

Abstract

The rapid increase in the Internet's connectivity has lead to proportional increase in the development of Web-based applications. Usage of downloadable content has proved effective in a number of emerging applications including electronic commerce, software components on-demand, and collaborative systems. In all these cases, Internet user agents (like browsers, tuners) are widely used by the clients to utilize and execute such downloadable content. With this new technology of using downloadable content comes the problem of the downloaded content obtaining unauthorized access to the client's resources. In effect, granting a hostile remote principal the requested access to client's resources may lead to undesirable consequences. Hence it is important for the browsers to provide a framework such that the user can fine tune his system according to his trust relationship with the content authors. Currently available systems either do not allow the downloaded content to access any of the local resources or allows all the contents to have the same privileges. In this paper, we present the design and implementation of a model that provides resource access control of a finer granularity for an user agent. Using our model, the client will be able to selectively grant access to resources based on a trust relationship with the principal, who has certified the authenticity of the contents.

1 Introduction

The ever-expanding nature of the Internet and the World Wide Web poses new problems such as scalability, standard naming scheme, and security. Nowadays it is becoming increasingly common to download some active content over the untrusted Internet and execute it on a client machine. This downloadable content can be Java [1] applets, Castanet [3] channel's contents or component objects like JavaBeans [2], and other executables. With the wide acceptance of object-oriented technology in every aspect of engineering, it is also common to envision all such content on the Web to be objects, accepting messages and providing the necessary services. Designing a scheme to protect client machines from hostile applets and components has become a necessity. Such a scheme should also provide the user the ability to selectively allow *trusted* contents to be downloaded and executed.

Protecting the client machine from hostile applets can be considered equivalent to providing a controlled access to a (client) system's resources. Devising such scheme calls for defining whom the client trusts, to identify the source of such downloadable content, verifying that the principal certifying the content (identity) is the same as it claims to be. This scheme should be flexible enough for the user to customize it to his security needs. The flexibility is now a requirement given the classifications of the network as Internet, corporate Intranet and Extranet (a domain consisting of an Intranet and multiple trusted client sub-domains). As more Intranet applications are developed it is common to assume that all such applications and downloadable content originating within the Intranet domain can be equally trusted. In this paper, we present the design and implementation of such a scheme for usage in user agents like browsers such that the restriction of access to resources in the name of high-security does not prohibit the users from using downloadable contents such as applets.

1.1 Motivation

The Internet has proved to be an effective data distribution medium, especially for software. The concept of downloadable content, where the software component (or the software itself) can be downloaded on-demand from the provider's (server) machine and executed on client machine, adds a flavor to this medium. From the point of view of software distributors, a new version of the software can just be installed on a server from where a client can download it or, in the case of Castanet [3], the tuner will automatically download updates in channels from transmitters. At the same time, a client can be assured of obtaining the latest version. The important aspect that affects such a developer-client

relationship over such an open medium like Internet is the varying trust relationships between any such pair. One way to protect client's resources is to prevent any (and all) downloadable content from accessing any of the client's resources. This is exactly the default policy enforced by the Java runtime to protect client machine from being attacked by all applets, the so called "sandbox security model." The user-agents (like browsers, tuners, etc.) incharge of downloading the content over the net and executing it on the client machine, widely adopt this default policy and hence restrict any applet from accessing the client's resources. Though this provides a solution for preventing hostile applets from attacking a client machine, its inflexibility prohibits the client to grant access to *trusted* applets. Hence, there is a high demand for a flexible mechanism for user agents (*browsers*) to serve the entire spectrum of trust relationship, varying from completely trusted Intranet contents to highly-untrusted Internet contents. Such a demand has been the motivation behind the modeling of the flexible system described in this paper.

1.2 Infrastructure

Our model is general enough to be applied to any environment of user agents and downloadable contents. Our implementation is tailored to the Java environment, as it is becoming the *de facto* standard for deploying Web-based applications. The basic infrastructure over which our system is built is the security framework of the JavaSoft's JDK1.1 release. We have taken advantage of digitally signed applets (which establishes an identity to base our trust on), the public-key key cryptography based mechanism for such exchange of contents, and the ACL (Access Control List) framework to associate a list of trusted identities with any object the client is trying to protect and Java's Sandbox security model.

The rest of the paper is organized as follows. Section 2 describes related work. An overview of the Java's sandbox security model is provided in Section 3. Definition of the components in our model is provided in Section 4. The specification details (identities, groups) required in our model is covered in Section 5. Description of our access control model is in section 6 followed by details of our trust policy over which we base our decision and the way in which we specify such policy, in section 7. Section 8 concludes this paper.

2 Related Work

Netscape Navigator 3.01 prohibits any applet downloaded over a network from accessing local files. Only those applets that reside on client machine which are accessible through the CLASSPATH (i.e. the content server is the client itself) can access the local files. The applets loaded over a network can reestablish network connections to only the site from which they were downloaded. Any other network connection to other sites is prohibited. This inflexible mechanism does not provide a way for users to fine tune their browser to allow trusted applets to access the local resources.

The HotJava1.0prebeta1 web browser provides a little flexibility to users in controlling accesses to local files. HotJava has encapsulated many parameters as *properties* ($< name, value >$ pairs) that can be configured by the user. These properties take effect when the browser is first invoked and changes to certain properties will be dynamically absorbed. Among those, properties of interest are the *acl.read* and *acl.write*. The value these properties take is a list of file (or directory) names. Specifying file (or directory) names indicate to the system that any applet run by the browser can read the files (or directories) listed in the *acl.read* and write to those listed under *acl.write* property. This means, either all applets read/write to a file/directory or none of the applets can.

Safe Tcl [4] consists of two interpreters: trusted interpreter and untrusted interpreter. The trusted interpreter provides access to the client machine's resources whereas the access is prohibited in the untrusted interpreter. The idea behind the SafeTcl is to run trusted code in trusted interpreter and untrusted one in the other. It lacks authentication and so all content is to be assumed to have been downloaded from untrusted sources.

The Telescript engine [5] uses *credentials* and *permits* for access control. The credentials establishes the identity of the principal responsible for the creation of the downloadable content. The permit is like a capability which grants access rights to other (including downloading client) principal's resources whereas the client can deny the right granted by the permit. Also permits do not have the scope of resource restrictions that we provide.

Cryptolopes (cryptographic envelopes) [6] provide a mechanism for protecting the content from hostile hosts. The client negotiates to access the content with the server. It helps providing security to the content whereas our system protects the client from the content.

Abadi et al. [7] present a calculus for access control from logical perspective. They have provided a logical language for access control lists and theories for making decisions on granting access requests. They have dealt with roles, by treating roles as a

composite principal which acts "as" the role (usually with reduced rights). In our system, we have not dealt with roles at this point. We plan to work on these extensions in future.

Jaeger et al. [10] describe an architecture for access control of downloaded content. Their architecture allows access of resources by downloaded content in a controlled manner. They map a remote principal to a principal group and determine the access rights. The four categories of principals they consider are: downloading principals, remote principals, applications developers and system administrator. Individual principals are aggregated into a principal group if they have the same rights. Such group rights are used to determine the rights of each individual principal. In our design, we define access to a resource using an ACL. This ACL is a set of $< principal, permissions >$ pairs. Thus in our method, same ACL can be used to define access rights for other resources. This increases the flexibility and reusability of ACL definitions.

3 Java Security Model

The implementation of our access control model is for a Java-enabled user agent like HotJava. Understanding the underlying security model is necessary to successfully augment advanced security features. As we are concerned with the security of a client system's resources against a downloaded content (applets), we will describe the Java's sandbox security model on which our implementation is based.

3.1 Security Reference Model Specifications

The Java Security Reference Model [13] defines an applet to be an executable Java program that is downloaded from the server. Also, applet loading and security is under the control of the application. Hence, defining our security policy for the browser is needed to provide necessary security enhancements as far as downloaded applets are concerned. The model defines a set of security interactions between Java components namely applet, application, Java virtual machine (JVM), client-server platforms and the server itself. Among those interactions, the *Applet Access Device Attempt(AADA)* is important to our model. According to AADA, an applet may attempt to call a method within the application (browser), such as an access to the local file system, or display. The application's Security Manager policy mediates the requested access. The invariant in the AADA is that the application always calls the Security Manager object to see if the requested access is permitted. This model (the sandbox model) in which access to resources goes through a security manager object of the application, helps to run untrusted code in a trusted environment and still ensure that the applet cannot damage the local machine.

3.2 Sandbox Security Model

Users can import and run applets from the Web or an intranet without damaging the client machine. Such an applet's actions are restricted to the "sandbox", which is an area dedicated by the browser to that applet. The applet cannot access any resources beyond the sandbox. This helps users ro run any (even untrusted) code and still ensure protection of their resources from attacks. The scope of such a sandbox is left to be defined by the browser. Our work presents a model to expand this sandbox to an extent user desires i.e., user should be able to define the scope of this sandbox depending upon the remote principals certifying downloadable contents.

According to the sandbox model, a security manager object serves as an access-approving authority within the application. Any attempts to access to resources, go through the security manager which in turn grants or denies access. The security manager is an object that is a subclass of the class *SecurityManager*. When an access is denied, the security manager throws a security exception.

Currently, the existing browsers don't have the flexibility to selectively allow access to selected applets. Our work fills this gap by defining a model to specify trust, resources that can be accessed on a per-applet basis, and necessary extension of the SecurityManager class to achieve the desired flexibility.

3.3 Establishing Trust

A mechanism to authenticate applets is necessary to define trust based on where the applet comes from. Digital signatures [11, 12] based on public-key cryptosystems come to the rescue. If an applet author can sign his applet, then the client can verify his signature and take necessary action: either deny or allow resource access requests. The JDK provides necessary framework for signing class files. To sign an applet, the author can bundle all Java code (class files) into a single Java archive file called a JAR file. Based on his private key and the contents of the JAR file, the author generates a digital signature block. On the client side, the security manager can resolve authentication issues by using the digital signature mechanism. Once the code is authenticated, then it can take the right decision based on user's access control specification.

Java provides the sandbox security model along with the mechanism for authentication using digital signatures. Our design is based on these available facilities in the Java security framework. In the next subsequent sections we will describe how we use this framework to help user specify trust, resource access control and how these specifications are interpreted by our security manager in effectively controlling access to the resources.

4 Model Components

Protecting client machines from malicious downloadable content is the objective of our system. Before we model a system that would achieve this, we need to define what we are trying to protect (resources) and from whom (principals certifying the contents). In this section we will define the granularity of such resources and varied categories of principals.

4.1 Resources

The resources on the client machine that need to be protected includes from files, directories on local disk, network connections, CPU usage, memory usage and access to the display. The resources can also be extended to non-physical components like remote objects, components like Java beans, etc. In our implementation, we will illustrate the protection of files and restricting network connections from Java applets. Effective application of remote objects [9, 8] may involve method invocations by downloadable content on objects residing in a client machine. In such cases, the user might be interested in protecting the object from being invoked or accessed by other hostile objects/applets.

4.2 Principals

When the issue of access control is raised, along with that comes the question of whom to trust. Implicitly there is an association of contents to some *principal* responsible for (creation or certification by digitally signing) that content. Such a principal can be another user, a company, a host, or a group of such entities. In an open distributed environment like the Internet, it is not impossible to impersonate other principals. This will lead to the user giving access to his resources to a principal, who is actually not whom he claims to be. Strong authentication is necessary in such an environment. The basic requirement for authentication is to define who principals are and how they can be authenticated.

In our model, principals can be individual users, companies, or hosts. With each of these principals, there should be a $< public, private >$ key pair associated. Using public key crypto techniques we can authenticate the principal. The notion of a principal can further be extended to groups of such principals. Assuming the existence of a name space to resolve identities of these principals, we can confirm their identities. Those principals can establish their identity along with the content they have developed, by signing them using digital signatures. We will describe the syntactic specification of resources and principals, as in our model, in the next section.

5 Specification of Principals

A standard format is required to specify principals and resources in any of the configuration files. Resources need to be specified only when its corresponding access control list (ACL) is configured. An ACL is a list of $< principal, permissions >$ pairs. Hence, principals need to be specified during formation of ACLs. Principals in an ACL can be individual users, hosts or group of these. Individual principals can be specified using their associated names (eg., *Nataraj, syrResearch, SyrUniv* or *diamondsTeam* for identities and *ratnam.cat.syr.edu or cat.syr.edu* for host name or domain name specification). Identity names are unique (we assume the existence of a global name space) and can be specified as such. Hence, a identity name can be name of individual identity like *Nataraj* or an identity of a team like *syrResearch* or a company or a body of companies and so on. In these cases, even though an identity might be a collection of other individual entities, it by itself is considered an identity. This notion of an entity representing a set of identities is different from *groups* of identities. A group is a set of identities sharing some common property. Each of those identities are called members of that group. A member can represent a group by signing for the group. But in the above case like *researchTeam*, though its a set of individual identities, it is a principal by itself having its own key pair. For such principals, other authorized principals can sign as the group.

Groups are sets of principals. Principals can be identities, hosts or other groups. Groups are specified separately in our system. They are specified in the format

```
<groupName>=<identityName>[,<identityName>]*
```

hence, following is an example of valid group specification:

```
syrResearch=Nataraj,Doug,Paul
diamondsTeam=Gary,Doug,Nataraj
syrHosts=cat.syr.edu,ece.syr.edu
catHosts=ratnam.cat.syr.edu,lynx.cat.syr.edu
```

The underlying *ACLParser* object (an instance of the sun.hotjava.security.ACLParser, responsible for parsing the ACLs specified in the predetermined format) parses this information and populates the *ACLManager* (an instance of the sun.hotjava.security.ACLManager, which maintains ACLs, policy database and the decision making authority for granting access). This is stored effectively like a lookup table as illustrated in Table 1. Any of the specified principals in an ACL should be registered in the user's identity database. An identity DB is created using the *javakey* utility of the JDK1.1. This utility helps specify the identity and how it is trusted and so on. This utility manages a database of entities (people, companies, etc.) and their keys (public and private) and certificates. This tool also generates signatures for JAR files and verifies those signatures [14]. It can be used by a client to declare whether or not it trusts certain entities.

The principals can also be specified using regular expressions. Hence, the following is also a valid specification.

```
allSyrHosts=*.syr.edu
```

Using a combination of regular expressions and other groups, new groups can be formed with ease. This makes the specification of principal groups easier.

6 ACL-based Model

Our model is based on associating access control list (ACL) with resources. We create named ACLs and associate them with resources. An ACL is associated with each resource to guard it and ACL itself is independent of the resource it guards. So a $< resource, ACL >$ pair means that the principals have the corresponding permissions on the resource. An ACL can be (re)used to guard more than one resource. The relationship between resources and ACLs are depicted in Figure 1. In this design, the key is the resource name. When a principal tries to access a resource, the system consults the configuration and obtains the ACL associated with the resource. It then checks the ACL to see if the principal under consideration has the required permission. If so, the system allows the principal to access the resource. Otherwise, it denies the attempted access.

In this system, the Java VM traps any access to the system's resources. The request is funneled through the security manager. The security manager is responsible for checking if the access is authorized by the user. The $< resource, ACL >$ association is formed during the start up of the application (which is executing downloaded content) and hence, its basically an associative lookup for ACL during the runtime. The user is also given the flexibility to add new $< resource, ACL >$ entries at runtime. The configuration is then dynamically updated and so is the database.

7 Trust Policy

In this section, we will describe the semantics of the decision to grant/deny access based on the specification of ACLs and policy. We will first understand the logic behind decisions taken by the security manager. We will then provide the format specification of ACLs and policies.

7.1 Access Granting Policy

Each user maintains a local security database containing the trust policy information. It consists of

- a database of principals (and keys) created using the *javakey* utility
- a specification of groups, formed by a set of principals
- a set of access control lists, containing $< principal, permissions >$ pairs
- a list of $< resource, ACL >$ (policy) pairs defining the trust relationship

The specification of the ACLs and associating resources with ACLs together form the trust policy database. When a request for an access to a resource is submitted, the application consults (through our enhanced security manager) this database to make a decision based on the ACL guarding the resource.

The Figure 2 depicts the decision flow in granting permission for a downloaded content to access a resource. The default policy is to deny any access unless the user explicitly grants access. If the identity is given access or denied access through explicit specification by the author, then the decision is based on that specification only. If there is no explicit individual specification of permission, the same check is carried out for all of the groups that the identity is a member of. Even if one of the groups is explicitly denied access, then the principal is denied access by the security manager. If all the groups are explicitly given access, then the principal is granted the request to access the resource. If no explicit permission is specified either as an individual identity or as a member of any of the group, then by default the access is denied. The browser then dynamically queries the user if he would like to allow the principal who has signed the applet to access the resource. Depending on user's input, the

GroupName	Principals	Principal Type
syrResearch	Nataraj, Doug, Paul	Identities
diamondsTeam	Gary,Doug,Nataraj	Identities
syrHosts	cat.syr.edu, ece.syr.edu	Hosts
catHosts	ratnam.cat.syr.edu,lynx.cat.syr.edu	Hosts

Table 1: A Group Table

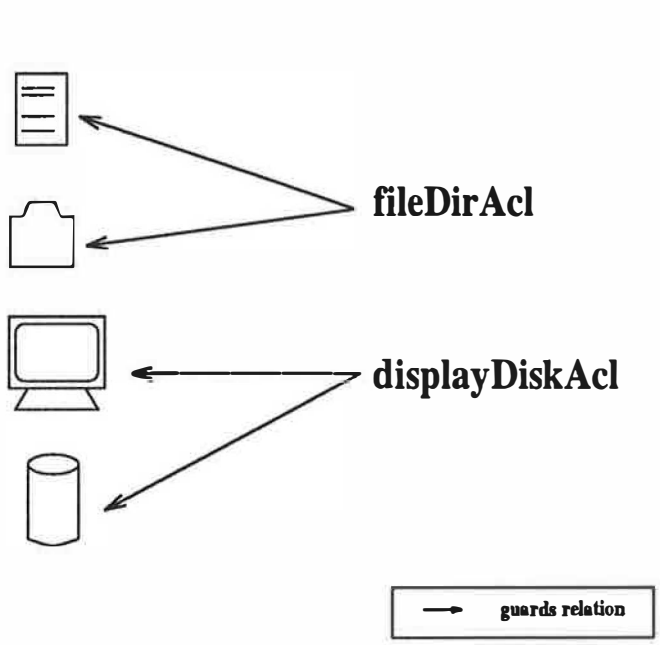


Figure 1: Sample Relation between Resources and ACLs

database as well as the runtime are dynamically updated. It is also possible for the user to specify negative permissions. This flexibility will be useful when specifying exceptions to group access permissions. For example, user might want to give access to all the members of a group except one member. In such a case, instead of forming a new group (a subgroup of original group), it is easier to grant permissions to the group and specify the member to be an exception. All combinations of principals can thus be accommodated in permission specification using the flexible format. The specification format for ACL and policy are given below.

7.2 ACL and Policy: Specification Format

An ACL relates principal to permissions. We use the ACL framework in JDK1.1 [14]. The principal is the key field in the ACL database. Given a principal name, one can obtain all the permissions associated with the principal. The format is as follows:

```
[+/-]{User|Group}.{Identity|Host}.
  <PrincipalName>=<setOfPermissions>
```

where,
the first field (optional) specifies whether the ACL specifies a granted permission or an exception. A – in that field indicates that it is an exception, i.e. the principal of given *PrincipalName* is explicitly denied the specified *setOfPermissions*. A + in the first field (or even if nothing is specified in that optional field) indicates that the principal is given the specified set of permissions. The key word *User*, in the second field, specifies that the principal name associated in this ACL is an individual principal whereas the keyword *Group* specifies that the principal name is actually the name of a group of principals. This resolves the *PrincipalName* specified in the fourth field to be either an individual principal or a group. The third field indicates that the specified principal

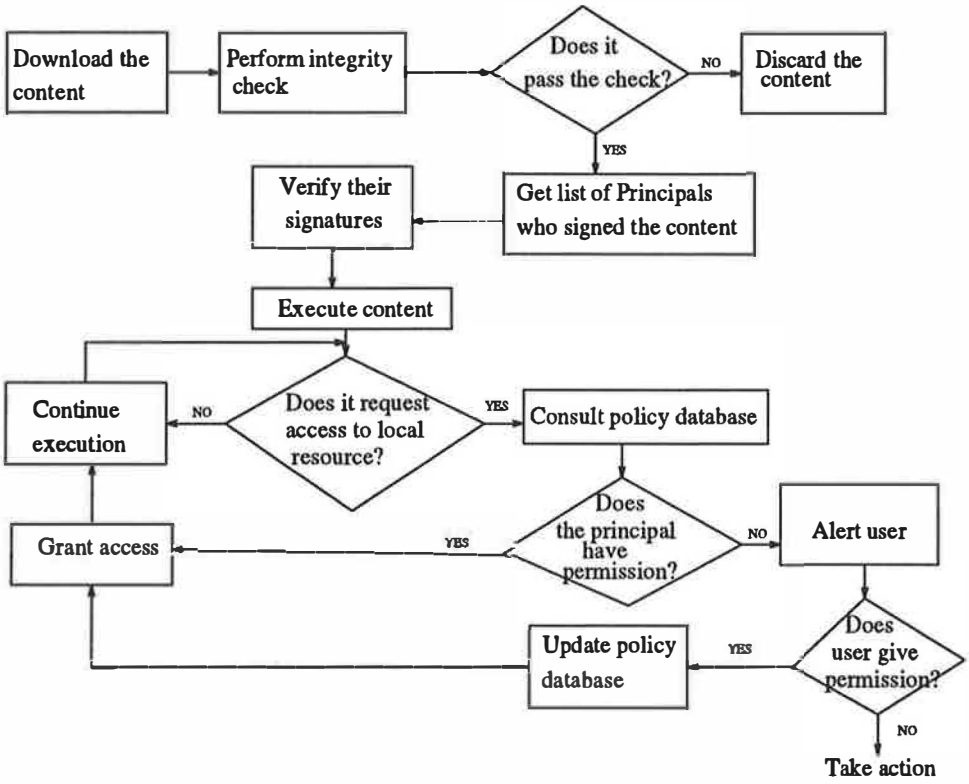


Figure 2: Decision flow for access control

is an Identity (name of a person, company, team, etc) or a Host (hostname, domainname, etc). The *setOfPermissions* is a list of permissions separated by a ','. Following example is a valid ACL specification.

```
+User.Identity.SyrUniv=FileRead, FileWrite
+Group.Host.catHosts=FileRead, FileWrite
-User.Host.ratnam.cat.syr.edu=FileWrite
```

In the above ACL (say it is named *acl1*), the principal SyrUniv has the FileRead and FileWrite permissions granted to it. All the host principals in syrHosts have the permission to FileRead and FileWrite except for the individual host cat.syr.edu (presumably a member of syrHosts group), which has been denied FileWrite permission. So effectively, the host cat.syr.edu has the permission FileRead through its membership in the syrHosts group but does not have the permission FileWrite even though all the other group members have the permission. For instance let the policy specification contain

```
/hostA/users/nataraj/javaWork/*=acl1
```

In this case, the ACL named acl1 guards the directory /hostA/users/nataraj/javaWork. So contents certified by SyrUniv can read and write to that directory. Also all the syrHosts can read and write to that directory with the exception of the host, ratnam.cat.syr.edu which cannot write to it.

8 Implementation Status

We have implemented our model for the HotJava browser. The browser allows selective access control of resources by the user. The user interface has been designed so that an end-user need not deal with the ACL or policy specification format details. The user can use the interface to specify the group and access configuration. The internal format and storage details are taken care of by our system.

We have built a security manager class BrowserSecurityManager which subclasses sun.applet.AppletSecurity. An instance of the BrowserSecurityManager is created when the browser is initialized. This security manager object has an ACLManager object which acts as an interface to the ACLs and the policy database. Whenever an access is attempted, the sandbox security model funnels the request to our security manager object. This object consults the ACLManager object to see if the remote principal (responsible for the applet which originates the access request) has the necessary permission(s) to perform the operation. On re-

Principals	Principal Type	Permissions	Permission Type
SyrUniv	Identity	FileRead, FileWrite	Grant
syrHosts	Hosts	FileRead, FileWrite	Grant
ratnam.cat.syr.edu	Hosts	FileWrite	Deny

Table 2: An ACL declaration

ceiving a *grant* message from the ACLManager object, the access is permitted. A security exception is thrown, otherwise.

The group and access configuration of the client system are read during initialization of the ACLManager object. With the assistance of ACLParser, the group specification and access specification are read and the ACLManager object is populated with this information.

We have provided implementations of the java.security.acl.Permission, java.security.acl.AclEntry and java.security.acl.Acl interfaces to serve our purpose. The BrowserAclImpl is responsible for both adding ACL entries (after reading the policy database) and for checking permissions during runtime. The class relationship is depicted in Figure 3 (for clarity, we have omitted method names from the class diagram).

The interface to the policy database is provided through the ACLManager object. In turn, the ACLParser object has the permission to read the policy database. The BrowserAclImpl object can add new entries at runtime (if the user wishes to add a trusted principal) and can update the policy database. Thus all these objects co-operate to control an access to client's resources. Currently, we have implemented our system to provide access control for the HotJava browser based on who has certified (signed) the applet and/or the source host of the applet (the host from where the applet is downloaded). In future, we plan to extend the communication through the Secure Socket Layer (SSL) and to provide flexible delegation mechanism, once the necessary delegation framework is in place.

9 Conclusion

We have presented a model to control the access to resources in an open distributed environment like the Internet. This model has been designed to provide advanced security features to user agents like browsers enabling them to selectively trust and grant access permissions to principals in such an open environment. This flexibility is critical not only for development of applications for the open untrusted Internet but for any trusted Intranet. The need for such a flexible security framework still exists. The public key cryptography techniques have been put to effective use in establishing authenticity over the network. Using our model, we have implemented a solution for the browsers to download contents over untrusted network and execute them in a trusted environment without damaging the client machines.

We have modeled our system with the flexibility of dealing with any kind of resource. Especially this will be useful with the current state of object-oriented technology where distributed objects cooperate to achieve their goal. Systems based on intelligent agents or distributed objects providing different services are being built. These objects may communicate either through remote method invocations like the RMI package [8] provides or by the another remote object mechanism proposed by us [9]. Given the Web's heterogeneous nature and Java's suitable positioning as an object-oriented platform for the Internet, extending our model to distributed objects can be easily done.

In a distributed environment, rights of a principal may be delegated to other principals. Thus the access control model needs to be extended to accommodate such delegated rights. As more distributed object-based systems are evolving and with speed in which Web-based applications are being deployed, the need for such a framework is necessary to provide a secure environment for remote accesses. In the future, we plan to develop a practical delegation model for secure distributed computation over the Internet.

References

[1] K. Arnold, and J. Gosling , "The Java Programming Language," *Addison-Wesley,* 1996

[2] "Java Beans: A Component Architecture for Java," *Online document at http://www.javasoft.com/beans,* December '96.

[3] "Castanet Whitepaper," *Online document at http://www.marimba.com,* October 1996.

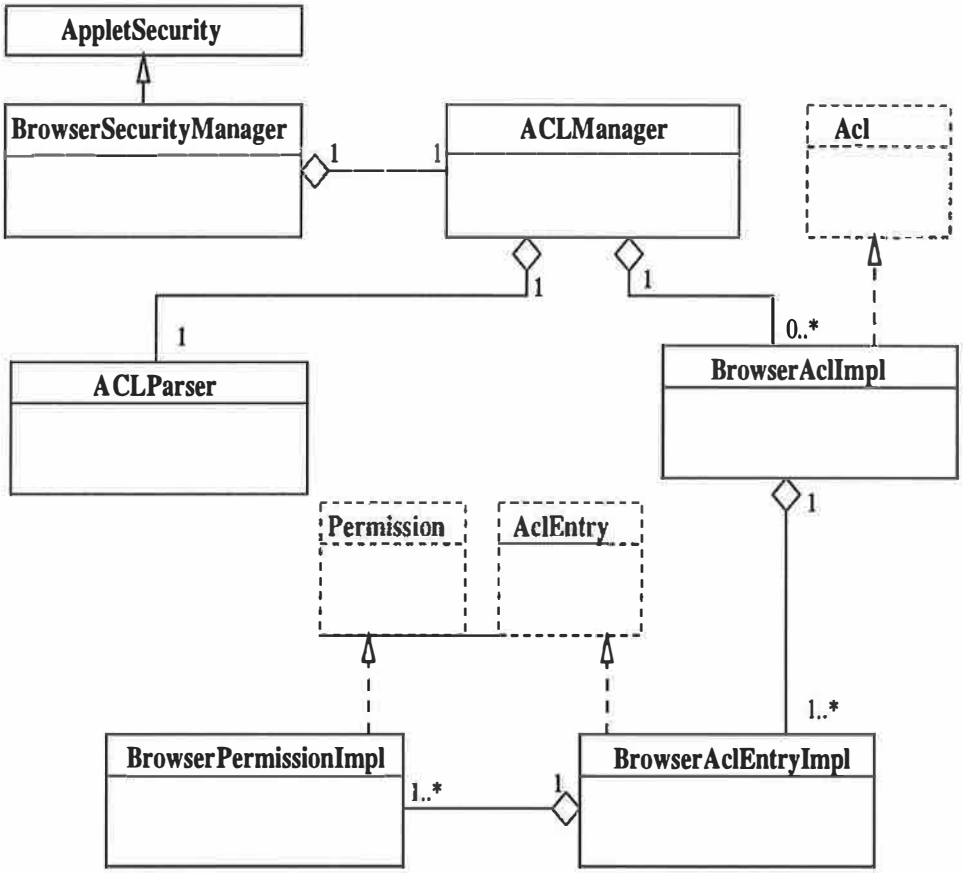


Figure 3: HotJava Resource Access Control Class Diagram

[4] J. Levy, and J. Ousterhout, "Safe Tcl: A Toolbox for Constructing Electronic Meeting Places," *First USENIX workshop on Electronic Commerce,* 1995.

[5] J. E. White, "Telescript Language Reference Manual," *General Magic Inc.,* October 1995.

[6] "Cryptolope Container Technology: A White Paper," *online documentation at http://www.cryptolope.ibm.com,* September 1996.

[7] M. Abadi, M. Burrows, and B. Lampson, "A Calculus for Access Control in Distributed Systems," *ACM Transactions on Programming Languages and Systems,* Vol 15 September 1993, pp706-734.

[8] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *Proceedings of the USENIX Conference on Object-Oriented Technology and Systems '96,* July 1996.

[9] N. Nagaratnam, A. Srinivasan, and D. Lea, "Remote Objects in Java," *Proceedings of the IASTED Intl. Conference on Networks,* January 1996.

[10] T. Jaeger, A. Rubin, and A. Prakash, "Building Systems That Flexibly Control Downloaded Executable Content", *Proceedings of the Sixth USENIX Security Symposium,* July 1996.

[11] B. Schneier, "Applied Cryptography," *Wiley and Sons,* 1994.

[12] R. Rivest, A. Shamir, and L. Adleman, "On Digital Signatures and Crypto Systems," *Communications of the ACM,* 1978.

[13] M. Erdos, B. Hartmann and M. Mueller, "Security Reference Model for the Java Developer's Kit 1.0.2,", *On-line document,* http://java.sun.com, 1997.

[14] Sun Microsystems, "Security in JDK1.1," *Java Documentation*, http://java.sun.com, 1997

Service Configurator

A Pattern for Dynamic Configuration of Services

Prashant Jain and Douglas C. Schmidt
pjain@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538[1]

Abstract

This paper describes the Service Configurator pattern, which decouples the implementation of services from the time when they are configured. This pattern increases the flexibility and extensibility of applications by enabling their constituent services to be configured at any point in time. The Service Configurator pattern is widely used in application environments (e.g., to configure Java applets into WWW browsers), operating systems (e.g., to configure device drivers), and distributed systems (e.g., to configure standard Internet communication services).

1 Introduction

A rapidly growing collection of services is now available on the Internet. The term *service* has several generally accepted meanings: (1) a single capability offered by a server (such as the `echo` service provided by the `inetd` superserver), (2) a collection of capabilities offered by a server (such as the `inetd` superserver itself), and (3) a collection of servers that cooperate to achieve a common task (such as a collection of `rwho` daemons in a LAN that periodically broadcast and receive status reports on user and host activities). Unless otherwise indicated, this paper uses the first definition of service, *i.e.,* an identifiable component in a server that offers a single capability to communicating entities.

The range of services available on the Internet include: WWW browsing and content retrieval services, software distribution services, electronic mail and network news transfer agents, file access on remote machines, remote terminal access, routing table management, host and user activity reporting, network time protocols, and object request brokerage services.

A common way to implement these services is to develop each one as a separate program and then compile, link, and execute each program in a separate process. However, this "static" approach to configuring services yields inflexible, and often inefficient, applications and software architectures. The main problem with this static approach is that it tightly couples the *implementation* of a particular service with the *configuration* of the service with respect to other services in an application or system.

This paper describes the *Service Configurator* pattern, which increases application flexibility, and often performance, by decoupling the behavior of services from the point in time at which these service implementations are configured into an application or system. The examples in this paper illustrate the Service Configurator pattern using Java applets. However, the Service Configurator pattern has been implemented in many ways, ranging from device drivers in modern operating systems (like Solaris and Windows NT) to Internet superservers (like `inetd` and the Windows NT Service Control Manager).

This paper is organized as follows: Section 2 describes the Service Configurator pattern using a variant of the GoF pattern format [1] and Section 3 presents concluding remarks.

2 The Service Configurator Pattern

2.1 Intent

Decouples the behavior of services from the point in time at which service implementations are configured into an application or system.

2.2 Also Known As

Super-server

2.3 Motivation

The Service Configurator pattern decouples the implementation of services from the time at which the services are configured into an application or a system. This decoupling improves modularity of the services and allows the services to evolve over time independently of configuration issues, such as whether or not two services must be co-located or what concurrency model will be used to execute the services.

In addition, the Service Configurator pattern provides centralized administration of all the services it configures. This facilitates automatic initialization and termination of the services and can optimize performance by performing common service initialization and termination activities.

[1]This research is supported in part by a grant from Siemens Medical Engineering.

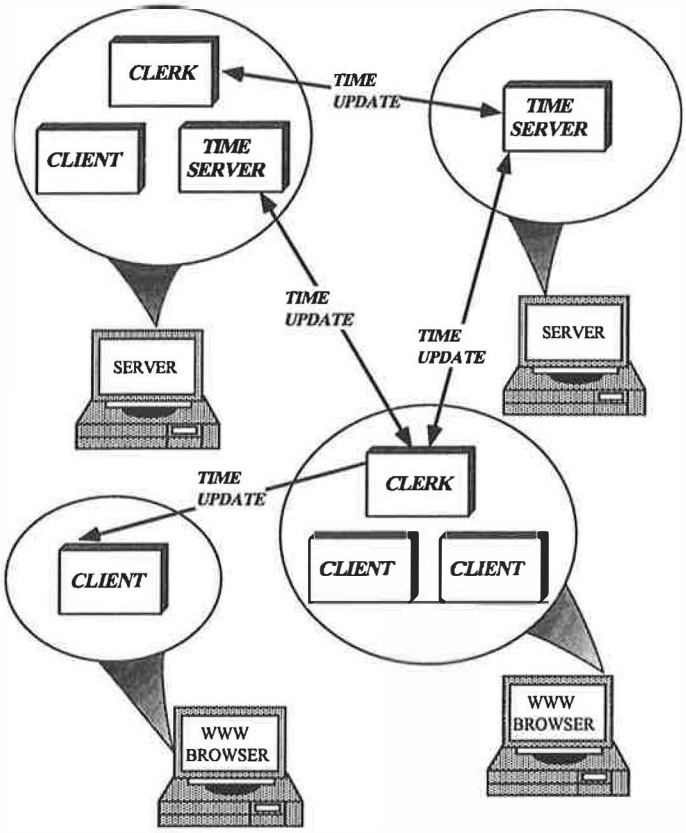


Figure 1: A Distributed Time Service

This section motivates the Service Configurator pattern using a distributed time service as an example.

2.3.1 Context

The Service Configurator pattern should be applied when a service needs to be initiated, suspended, resumed, and terminated dynamically. In addition, the Service Configurator pattern should be applied when service configuration decisions must be deferred until run-time.

To illustrate this pattern, consider the distributed time service shown in Figure 1. This service provides accurate, fault-tolerant clock synchronization for computers collaborating in local area networks and wide area networks. A synchronized time service is important in distributed systems that require multiple hosts to maintain accurate global time. For instance, large-scale distributed medical imaging systems [2] require globally synchronized clocks to ensure that patient exams are accurately timestamped and analyzed expeditiously by radiologists throughout the health-care delivery system.

As shown in Figure 1, the architecture of the distributed time service contains the following components:

- *Time Server* – which answers queries about the time made by Clerks.
- *Clerk* – which queries one or more Time Servers to determine the correct time, calculates the approximate correct time using one of several distributed time algorithms [3, 4], and updates its own local system time.
- *Client* – which uses the global time information maintained by a Clerk to provide consistency with the notion of time used by clients on other hosts.

2.3.2 Common Traps and Pitfalls

One way to implement the distributed time service is to statically configure the *logical* functionality of Time Servers, Clerks, and Clients into separate *physical* stand-alone processes. In this static approach, one or more hosts would run Time Server processes, which handle time update requests from Clerk processes. Each host that requires global time synchronization would run a Clerk process. The Clerks periodically update their local system time based on values received from one or more Time Servers. Client processes would then use the synchronized time reported by their local Clerk.[2]

In addition to the time service, other services (such as file transfer, remote login, and HTTP servers) provided by the hosts would also execute in separate statically configured processes.

However, implementing and configuring services in the static manner shown above has the following drawbacks:

• **Service configuration decisions must be made too early in the development cycle:** This early binding is undesirable since developers may not know *a priori* the best way to co-locate or distribute service components. For example, minimal memory resources in wireless computing environments may force the split of Client and Clerk into two independent processes running on separate hosts. In contrast, in a real-time avionics environment it might be necessary to co-locate the Clerk and the Time Server into one process to reduce communication latency. Forcing developers to commit prematurely to a particular service configuration impedes flexibility and can reduce performance and functionality.

• **Modifying or terminating a service may adversely affect other services:** In the static approach, the implementation of each service component is tightly coupled with its initial configuration. This makes it hard to modify one service without affecting other services. For example, in the real-time avionics environment mentioned above, a Clerk and a Time Server might be statically configured to execute in one process to reduce latency. If the distributed time algorithm implemented by the Clerk is changed, the existing Clerk code would require modification, recompilation, and relinking. However, terminating the process to change the Clerk code would also terminate the Time Server. This disruption in service availability may not be acceptable for mission critical distributed systems (such as telecommunication switches or call centers [5]).

[2]For platforms that support shared memory, communication overhead can be minimized by storing the current time into shared memory that is mapped into the address space of the Clerk and all Clients on the same host.

• **System performance may not scale up efficiently:** Associating a process for each service ties up valuable OS resources (such as I/O descriptors, virtual memory, and process table slots). This can be wasteful if services are frequently idle. Moreover, processes are often the wrong concurrency model for many short-lived communication tasks (such as asking a Time Server for the current time or resolving a host address request in the Domain Name Service). In these cases, a multi-threaded Active Object [6] or a single-threaded Reactive [7] event loop may be more efficient.

2.3.3 Solution

Often, a more convenient and flexible way to implement distributed services is to use the *Service Configurator pattern*. This pattern decouples the behavior of services from the point in time at which the service implementations are configured into an application or system. The Service Configurator pattern resolves the following forces:

• **The need to defer the selection of a particular type, or a particular implementation, of a service until very late in the design cycle:** Dynamic configuration allows developers to concentrate on the functionality of a service, without committing themselves prematurely to a particular configuration of services. By decoupling functionality from configuration, the Service Configurator pattern permits applications to evolve independently of the configuration policies and mechanisms used by the system.

• **The need to build complete applications or systems by composing multiple independently developed services that do not require global knowledge:** The Service Configurator pattern requires all services to have a uniform interface for configuration and control. This allows the services to be treated as modular building blocks that can be integrated easily as components in a larger application. Enforcing a uniform interface for all services makes them "look and feel" the same with respect to how they are configured, thereby simplifying application development.

• **The need to optimize and control the behavior of a service at run-time:** Decoupling implementation details of a service from configuration decisions makes it possible to fine-tune certain implementation or configuration parameters of services. For instance, depending on the parallelism available on the hardware and operating system, it may be either more or less efficient to run one or more services in separate threads or processes. The Service Configurator pattern enables applications to select and tune these behaviors at run-time, when additional information (such as the number of CPUs or the OS version) is available to help optimize the services. In addition, adding a new or updated service to a distributed system may not require downtime for existing services.

Figure 2 uses OMT notation to illustrate the structure of the distributed time service designed according to the Service Configurator pattern.

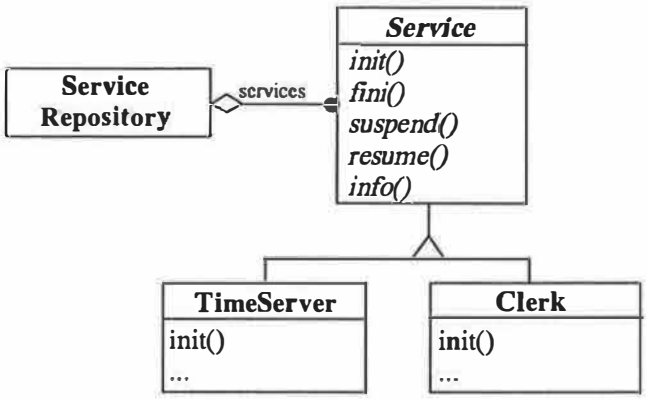


Figure 2: Structure of a Distributed Time Service

The `Service` base class provides a standard interface for configuring and controlling services (such as Time Servers or Clerks). A Service Configurator-based application uses this interface to initiate, suspend, resume, and terminate a service, as well as to obtain service-specific information (such as the service name, host address, and port number). Services reside within a `Service Repository` and can be added to and removed from the `Service Repository` by Service Configurator-based applications.

Two subclasses of the `Service` base class appear in the distributed time service: `TimeServer` and `Clerk`. Each subclass represents a concrete `Service`, which has specific functionality in the distributed time service. The `TimeServer` service is responsible for receiving and processing requests for time updates from `Clerks`. The `Clerk` service is a Connector [8] factory that performs the following tasks:

1. Creates a new connection for every server;
2. Dynamically allocates a new handler to send time update requests to a connected server;
3. Receives the replies from all the servers through the handlers;
4. Updates the local system time based on an average of all `TimeServer` responses.

The Service Configurator pattern improves the flexibility of the distributed time service by managing the configuration of service components in the time service. Thus, configuration decisions (such as whether or not to co-locate the `TimeServer` and `Clerks`) are decoupled from implementation details (such as the algorithm used by a Clerk to update its notion of time). In addition, implementations of the Service Configurator pattern can provide a framework that consolidates the configuration and management of application services in one administrative unit.

2.4 Applicability

Use the Service Configurator pattern when:

- Services must be initiated, suspended, resumed, and terminated dynamically; and
- An application or system can be simplified by being composed of multiple independently developed and dynamically configurable services; or
- The management of multiple services can be simplified or optimized by configuring them using a single administrative unit.

Do *not* use the Service Configurator pattern when:

- Dynamic configuration is undesirable due to security restrictions (in this case, static configuration of trusted services may be necessary); or
- The initialization or termination of a service is too complicated or too tightly coupled to its context to be performed in a uniform manner; or
- Stringent performance requirements mandate the need to minimize the extra levels of indirection used by the the OS and language mechanisms for dynamic configuration.

2.5 Structure and Participants

The structure of the Service Configurator pattern is illustrated using OMT notation in Figure 3.

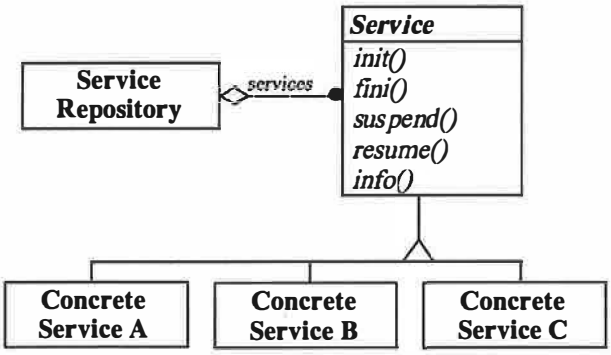


Figure 3: Structure of the Service Configurator Pattern

The key participants in the Service Configurator pattern include the following:

- **Service** (`Service`)
  - Specifies the interface that contains the abstract *hook methods* [9] (such as methods for initialization and termination) used by a Service Configurator-based application to dynamically configure each `Service`.
- **Concrete Service** (`Clerk` and `TimeServer`)
  - Implements the service hook methods and other service-specific functionality (such as event processing and communication with clients and other services).
- **Service Repository** (`ServiceRepository`)
  - Maintains a repository of all services offered by a Service Configurator-based application. This allows an administrative entity to centrally manage and control the behavior of application services.

2.6 Collaborations

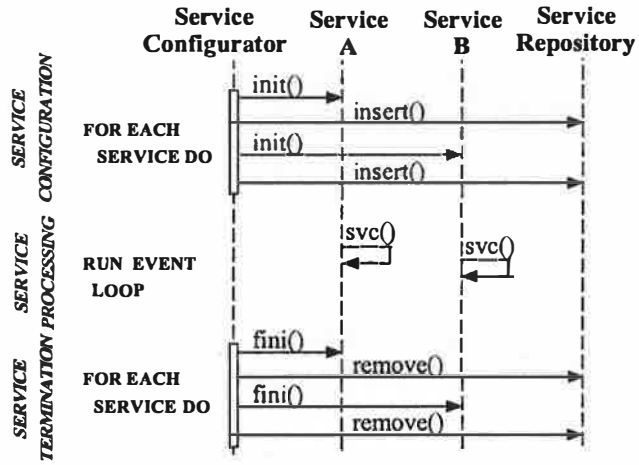


Figure 4: Interaction Diagram for the Service Configurator Pattern

Figure 4 depicts the collaborations in following three phases between components of the Service Configurator pattern:

- *Service configuration* – The Service Configurator initializes a `Service` by calling its `init` method. Once the `Service` has been initialized successfully, the Service Configurator adds it to the `ServiceRepository`. The `ServiceRepository` is used by the Service Configurator to manage and control all `Services` that are installed.
- *Service processing* – A `Service` is executed once it has been configured into the system. Once a `Service` is executing, the Service Configurator can suspend and resume the `Service`.
- *Service termination* – The Service Configurator terminates a `Service` once it is no longer needed. The Service Configurator calls the `fini` method on the `Service` to allow it to clean up before terminating. Once a `Service` is terminated, it is removed from the `ServiceRepository`.[3]

[3]Not all systems support service termination. For example, the Java run-time environment that implements the Service Configurator pattern provides no way to terminate an applet or unload it once it has been loaded into the run-time environment (*e.g.*, a WWW browser).

2.7 Consequences

2.7.1 Benefits

The Service Configurator pattern offers the following benefits:

• **Centralized administration of services:** The pattern consolidates one or more services into a single administrative unit. This simplifies development by automatically performing common service initialization and termination activities (such as opening and closing files, acquiring and releasing locks, etc.). In addition, it centralizes the administration of services by enforcing a uniform set of configuration management operations on them (such as *initialize*, *suspend*, *resume*, and *terminate*).

• **Increased modularity and reuse:** The pattern improves the modularity and reusability of services by decoupling the *implementation* of these services from the *configuration* of the services. In addition, all services have a uniform interface by which they are configured, thereby encouraging reuse and simplifying development of subsequent services.

• **Increased opportunity for tuning and optimization:** The pattern increases the range of service configuration alternatives available to developers by decoupling service functionality from the concurrency models (*e.g.*, threads or processes) used to invoke the service. Developers can adaptively tune daemon concurrency levels to match client demands and available OS processing resources by choosing from a range of concurrency models. Some alternatives include spawning a thread or process upon receipt of a client request or pre-spawning a thread or process at service creation time.

2.7.2 Drawbacks

The Service Configurator pattern has the following drawbacks:

• **Lack of determinism:** The pattern makes it hard to determine the behavior of a service and/or application until run-time. This is particularly problematic for real-time systems since a dynamically configured service may perform unpredictably when run with certain services. For example, if consumers in a real-time Event Service do not obey their periodic processing constraints, other real-time services will miss their deadlines and the system will not behave predictably.

• **Reduced reliability:** An application that uses the Service Configurator pattern may be less reliable than one that does not because a particular configuration of services may adversely affect the execution of the services. For instance, a faulty service may crash, thereby corrupting state information it shares with other co-located services. This is particularly problematic for open systems [10], such as Java applets within WWW browsers that configure and execute multiple services within the same process.

• **Increased overhead:** The pattern adds extra levels of indirection to execute a service. For instance, the Service Configurator first initializes the service and then loads it into the `Service Repository`. This may be undesirable or an unnecessary overhead in time-critical applications. In addition, the Service Configurator pattern often configures services via dynamic linking, which adds extra indirection to invoke functions and access global variables [11].

• **Lack of generality:** If services are tightly coupled, it may not be possible to dynamically configure them in arbitrary ways using the Service Configurator pattern. For example, it may be necessary to configure two services in a specific order or it may be necessary to always co-locate two services. The Service Configurator pattern only provides the mechanism of decoupling service implementation from service configuration – it does *not* dictate any policy by which services are to be configured. Therefore, the Service Configurator is a building block in a "pattern language" of strategies for dynamically configuring and reconfiguring services.

2.8 Implementation

The Service Configurator pattern has been implemented in many contexts, ranging from device drivers in operating systems like Solaris and Windows NT, Internet superservers like inetd, and Java applets in WWW browsers. This section explains the steps and alternatives involved when implementing the pattern. These steps and alternatives are summarized in Table 1.

• **Define the service control interface:** The following is the core interface that a service should support to enable the Service Configurator to configure and control the service:

- *Initialization* – Provides an entry point into the service and performs initialization of the service;
- *Termination* – Terminates execution of a service and provides a hook to cleanup application resources;
- *Suspension* – Temporarily suspends the execution of a service;
- *Resumption* – Resumes execution of a suspended service;
- *Information* – Obtains information about a service to determine its identity and behavioral characteristics.

There are two ways to define the service control interface: *inheritance-based* and *message-based*, as described below:

- *Inheritance-based service control interface* – In this approach, each service inherits from a common base class. This approach is used by the ACE `Service Configurator` framework [5] and Java applets, which defines abstract base classes that contain pure virtual "hook" methods. The following shows the `Service` interface similar to the one provided in ACE:

```
class Service
{
```

Step	Common Alternatives
Define the service control interface	• Services inherit from an abstract base class • Services respond to control messages
Define a Service Repository	• Maintain an in-memory table of service implementations • Maintain a persistent database of service implementations
Select a configuration mechanism	• Specify at command line • Specify through a configuration file • Specify through a user interface
Determine service execution mechanism	• Reactive execution • Multi-threaded Active Objects • Multi-process Active Objects

Table 1: Steps Involved in Implementing the Service Configurator Pattern

```
public:
  // = Initialization and termination hooks.
  virtual int init (int argc, char *argv[]) = 0;
  virtual int fini (void) = 0;

  // = Scheduling hooks.
  virtual int suspend (void);
  virtual int resume (void);

  // = Informational hook.
  virtual int info (char **, size_t) = 0;
};
```

The `init` method is the entry point hook into a `Service`. It is used by the Service Configurator to initialize and execute a `Service`. The `fini` method is a hook that allows the Service Configurator to terminate the execution of a `Service`. The `suspend` and `resume` methods serve as scheduling hooks and are used by the Service Configurator to suspend and subsequently resume the execution of a `Service`. The `info` method allows the Service Configurator to obtain `Service`-related information (such as its name and network address). Together, these methods impose a contract between the Service Configurator and the `Service` objects that it manages.

- *Message-based service control interface* – Another way to control services is to program them to respond to a specific set of messages. This makes it possible to integrate the Service Configurator into non-OO programming languages (such as C). The Windows NT Service Control Manager (SCM) [12] uses this scheme. Each Windows NT host has a master SCM process that automatically initiates and manages system services by passing them control messages such as PAUSE, RESUME, and TERMINATE. Each developer of an SCM-managed service must write code to process these messages and perform the intended actions.

• **Define a Repository:** A `Service Repository` maintains all the `Service` implementations in the form of objects, executable programs, and/or dynamically linked library (DLLs). A Service Configurator uses the `Service Repository` to access a service when it is configured into or removed from the system. In addition, the Repository maintains the current status of each service (*e.g.*, whether a service is active or suspended). This information may reside in main memory, a file system, or the kernel.

• **Select a configuration mechanism:** A service must be configured before it can execute. Configuring a service requires specifying attributes that indicate the location of the service's implementation (such as an executable program or DLL), as well as the parameters required to initialize a service at run-time. This configuration information can be specified in various ways (*e.g.*, on the command line, through a user interface, or through a configuration file). A centralized configuration mechanism (such as the NT Registry or `inetd.conf` file) simplifies the installation and administration of the services in an application by consolidating service attributes and initialization parameters in a single location.

• **Determine the service concurrency model:** A service that has been dynamically configured by a Service Configurator can be executed using various combinations of Reactive [7] and Active Object [6] schemes. These alternatives are briefly outlined below:

- *Reactive execution* - This approach uses a single thread of control to execute the Service Configurator and all the services it configures.
- *Multi-threaded Active Objects* - This approach runs the dynamically configured services in their own threads of control within the Service Configurator process. The Service Configurator can either spawn new threads "on-demand" or execute the services within an existing pool of threads.
- *Multi-process Active Objects* - This approach runs the dynamically configured services in their own processes. The Service Configurator can either spawn new processes "on-demand" or execute the services within an existing pool of processes.

2.9 Sample Code

The following code shows an example of the Service Configurator pattern in the context of Java *applets*. An applet is a Java class that can be loaded and run by a Java application (such as a Web browser, an applet viewer, or an application). The example below focuses on the configuration-related aspects of the distributed time service described in Section 2.3. In addition, this example illustrates how other patterns (such as the Active Object pattern [6] and the Acceptor and Connector patterns [8]) are commonly used in conjunction with the Service Configurator pattern to develop flexible communication infrastructure and services.

In the example, the `Concrete Service` class in the OMT class diagram shown in Figure 3 is represented by the `TimeServer` class and the `Clerk` class. The Java code in this section implements the `TimeServer`

and the Clerk classes.[4] Both classes inherit from java.applet.Applet. This allows them to be downloaded (*e.g.*, from an HTTP server) and dynamically configured (*e.g.*, into a Java interpreter within a WWW browser).

The WWW server's file system serves as the Service Repository for the Java applets. In addition, the java.applet.Applet class provides hook methods that allow dynamic (1) configuration of a service (init), (2) suspension of a service (stop), and (3) resumption of a service (start). Note that the java.applet.Applet class does not provide a termination method equivalent of fini described in Section 2.8. The Service Configurator pattern remains at the heart of the Java applets, however, by allowing their implementation to be decoupled from their dynamic configuration.

2.9.1 The TimeServer Class

The TimeServer uses the Acceptor class to accept connections from one or more Clerks. The Acceptor class uses the Acceptor pattern [8] to create handlers for each Clerk connection that wants to receive requests for time updates [13]. This design decouples the implementation of the TimeServer from its configuration. Therefore, developers can change the implementation of the TimeServer independently of its configuration. This design provides flexibility with respect to evolving the implementation of the TimeServer class.

The TimeServer class inherits from the standard java.applet.Applet class, which enables a TimeServer to be dynamically loaded into a running Java application. Once the TimeServer applet has been downloaded and verified, the Java run-time system invokes its init hook. This method performs the Time Server-specific initialization code.

The TimeServer class implements the Runnable interface. This allows it to become an active object and run in its own thread of control. Running TimeServer as an active object is useful if the applet's main thread of control must perform other tasks (such as responding to user GUI events and methods called by the system).

```
import java.applet.Applet;

public class TimeServer extends Applet
                        implements Runnable
{
  // Initialize the TimeServer when loaded.  This
  // may include synchronizing server clock with
  // an atomic clock.  This method corresponds
  // to the init() hook method of the Service
  // Configurator pattern.

  public void init ()
  {
    // Initialize.
  }

  // (Re)start the TimeServer.  Note that this method
  // gets called after init() when the applet first
  // starts up in the context of Java run-time
  // system and also when the applet is restarted
  // after being temporarily stopped.  The method
  // spawns off a new thread to handle Clerk
  // connections if a thread is not already running.
  // Otherwise it resumes the currently suspended
  // thread.  This method corresponds to the resume()
  // hook method of the Service Configurator pattern.

  public void start ()
  {
    if (serverThread_ == null) {
      serverThread_ = new Thread (this);
      serverThread_.start ();
    }
    else
      // Resume the server thread.
      serverThread_.resume ();
  }

  // Temporarily stop/suspend the TimeServer.
  // This method suspends the thread that handles
  // Clerk connections.  This method corresponds
  // to the suspend() hook method of the Service
  // Configurator pattern.

  public void stop ()
  {
    if (serverThread_ != null &&
        serverThread_.isAlive ()) {
      // Suspend the server thread.
      serverThread_.suspend ();
    }
  }

  // Return information about the TimeServer
  // by overriding the method defined in the
  // java.applet.Applet class.  This method
  // corresponds to the info() hook method of
  // the Service Configurator pattern.
  public String getAppletInfo ()
  {
    // Return a String containing information
    // about this applet. This may include the
    // name of the host, the version number, etc.
    return new String ( ... );
  }

  // This method serves as the entry point for
  // the Time Server thread.  It is called
  // when the thread starts.
  public void run ()
  {
    // Set the connection acceptor_ endpoint into
    // listen mode (using the Acceptor pattern).
    acceptor_.open (port_);

    // Now use the acceptor_ to accept
    // connections from Clerks.
    // ...
  }

  // Acceptor used for Clerk connections.
  protected Acceptor acceptor_;

  // Port the TimeServer listens on.
  private int port_ = SERVER_PORT_NUMBER;

  // The Server Thread
  private Thread serverThread_ = null;

  // ...
}
```

The Java run-time system can suspend and resume the TimeServer by calling its stop and start hooks, respectively. In addition, it can call getAppletInfo method to obtain useful information about the service, such as the version number or the name of the author.

[4] To save space, most of the detailed Java code and exception handling code has been omitted.

2.9.2 The Clerk Class

The `Clerk` uses the Connector pattern [8] to establish and maintain connections with one or more `TimeServers`. The Connector pattern creates a handler for every connection to a TimeServer. The handlers receive and process time updates from the `TimeServers`.

The `java.applet.Applet` base class is the parent of the `Clerk` class. Therefore, like the `TimeServer`, a `Clerk` can be dynamically configured by the Java run-time system acting in the role of Service Configurator. The Java run-time system can initialize, suspend, resume, and obtain information about the `Clerk` by calling its `init`, `stop`, `start`, and `getAppletInfo` hooks, respectively.

```
import java.applet.Applet;

public class Clerk extends Applet
                   implements Runnable
{
  // Initialize the Clerk when loaded.  This
  // may include initializing the algorithm
  // implementation to be used to compute the
  // Clerk's notion of time.  This method
  // corresponds to the init() hook method of
  // the Service Configurator pattern.
  public void init ()
  {
    // Initialize.
  }

  // (Re)start the Clerk.  Note that this method
  // gets called after init() when the applet first
  // starts up in the context of Java run-time
  // system and also when the applet is restarted
  // after being temporarily stopped. The method
  // spawns off a new thread to setup connections
  // with the TimeServers if a thread is not already
  // running. Otherwise it resumes the currently
  // suspended thread.  This method corresponds to
  // the resume() hook method of the Service
  // Configurator pattern.

  public void start ()
  {
    if (clerkThread_ == null) {
      clerkThread_ = new Thread (this);
      clerkThread_.start ();
    }
    else
      // Resume the Clerk thread.
      clerkThread_.resume ();
  }

  // Temporarily stop/suspend the Clerk.  This
  // method suspends the thread that handles
  // connection to TimeServers.  This method
  // corresponds to the suspend() hook method
  // of the Service Configurator pattern.

  public void stop ()
  {
    if (clerkThread_ != null &&
        clerkThread_.isAlive ()) {
      // Suspend the Clerk thread.
      clerkThread_.suspend ();
    }
  }

  // Return information about the Clerk by
  // overriding the method defined in the
  // java.applet.Applet class.  This method
  // corresponds to the info() hook method of
  // the Service Configurator pattern.
  public String getAppletInfo ()
  {
    // Return a String containing information about
    // this applet. This may include the name of
    // the host, the version number, etc.
    return new String ( ... );
  }

  // This method serves as the entry point for
  // the Clerk thread. It is called when the
  // thread starts.
  public void run ()
  {
    // Use the connector to set up connections
    // to all the TimeServers.  Then use the
    // updateTime() method to send periodic requests
    // to the TimeServers for time updates, receive
    // the requests from the TimeServers, and compute
    // the local notion of time.
    // ...
  }

  // Called periodically to compute the local
  // system time.
  protected void updateTime (long t)
  {
    // Implement Clock Synchronization algorithm
    // here to compute local system time.
  }

  // Connect to TimeServers.
  protected Connector connector_;

  // The Clerk Thread.
  private Thread clerkThread_ = null;
}
```

The `Clerk` periodically sends a request for time update to all its connected `TimeServers`. Once the `Clerk` receives responses from all its connected `TimeServers`, it recalculates its notion of the local system time. Thus, when Clients ask a Clerk for the current time, they receive a globally synchronized value.

2.9.3 Lifecycle of a Service

Figure 5 shows a state diagram of the lifecycle of a `Service` (such as the Clerk service).

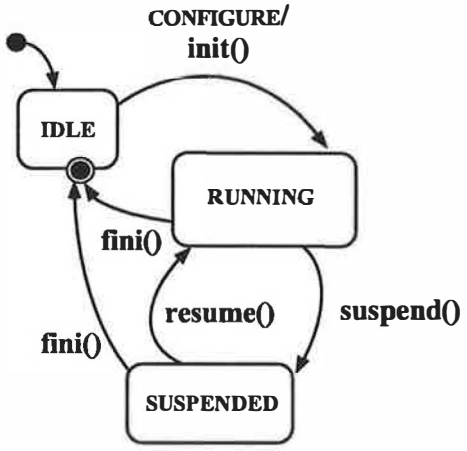


Figure 5: State Diagram of a Service Lifecycle in the Service Configurator Pattern

Initially the service is idle. Depending upon requirements, the user can choose from various implementations dynamically, without having to focus on configuration issues.

For instance, different Clerk services may exist corresponding to different algorithm implementations. Thus, the user may either select a Clerk service that implements the Berkeley algorithm [3] or a Clerk service that implements Cristian's algorithm [4]. The choice may depend upon the characteristics of the `TimeServer`. If the machine on which the `TimeServer` resides has a WWV receiver[5] the `TimeServer` can act as a passive entity and Cristian algorithm would be best suited. On the other hand, if the machine on which the `Time Server` resides does not have a WWV receiver then an implementation of the Berkeley algorithm would be more appropriate.

Once a Clerk service has been selected, it can be easily configured by loading it into the Java run-time environment (such as a Web browser, an applet viewer, or an application). The following HTML fragment shows how the Clerk applet can be loaded in an applet viewer or a Web browser:

```
<APPLET code="Clerk.class">
<PARAM name=configFile value="svc.conf">
<PARAM name=pollTime value="10">
</APPLET>
```

The `APPLET` tag specifies an applet to be run within a Web browser or an applet viewer. The `PARAM` tag specifies named parameters to be passed to the applet. An applet can look up the value of a parameter specified in a `PARAM` tag with the `Applet.getParameter` method. In the example above, the Clerk applet is passed the name of a service configuration file and a `pollTime` of 10 seconds. This configuration file (`svc.conf`) contains the hostnames and port numbers of all the Time Servers the Clerk will connect to. The `pollTime` indicates how frequently the Clerk will poll the Time Servers.

To reduce communication latency, The Clerk service can be co-located with a Time Server service. The following HTML fragment shows how the Clerk applet can be loaded in an applet viewer or a Web browser together with a co-located Time Server applet:

```
<APPLET code="Clerk.class">
<PARAM name=configFile value="svc.conf">
<PARAM name=pollTime value="10">
</APPLET>
<APPLET code="Server.class">
<PARAM name=port value="7734">
</APPLET>
```

In this example, the Time Server class will listen at port 7734.

Figure 6 shows the Clerk running independently as well as running co-located with a Time Server. This configuration decision need not affect the implementation of the various time services. Note, however, that if the Clerk and the Time Server are co-located in the same process, the Clerk may optimize communication by (1) eliminating the need to set up a communication channel with the Server and (2) directly accessing the Server's local notion of time via shared memory. In general, the decoupling between a service implementation and its configuration exemplifies the flexibility offered by the Service Configurator pattern.

[5] A WWV receiver intercepts the short pulses broadcasted by the National Institute of Standard Time (NIST) to provide Universal Coordinated Time (UTC) to the public.

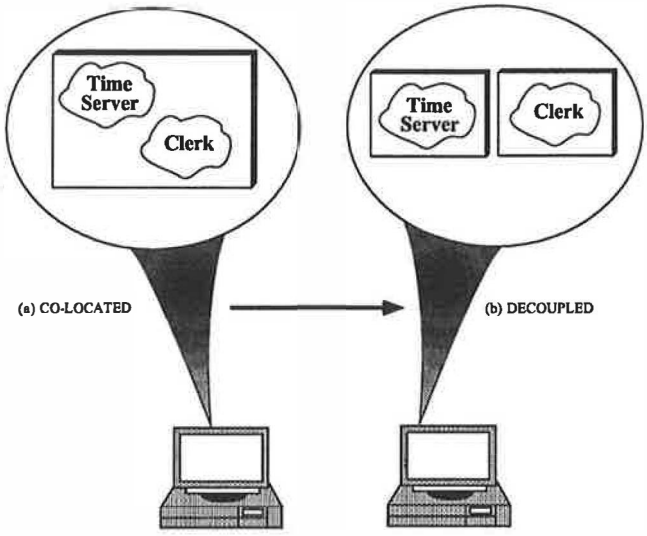


Figure 6: (a) Clerk co-located with a Time Server; (b) Clerk running independently

2.10 Known Uses

The Service Configurator pattern has been used in a wide range of operating system and application programming environments including Java applets, UNIX, Windows NT, and ACE:

• **Java applets:** The applet mechanism in Java uses the Service Configurator pattern. Java supports dynamically downloading, initializing, starting, suspending and resuming applets. For instance, it defines methods (*e.g.*, `stop` and `start`) that suspend and resume applet threads. A method in a Java applet can access the thread it is running in using `Thread.currentThread()`. In addition, threads can control each other by invoking methods like `stop` and `start`.

• **Modern operating system device drivers:** Modern operating systems (such as Solaris [14] and Windows NT [12]) support dynamically configurable kernel-level device drivers. For instance, Solaris drivers can be linked into and unlinked out of the system dynamically via `init/fini/info` hooks. This makes it possible to reconfigure the operating system without having to shut it down, recompile and relink new drivers into the kernel, and then restart the system.

• **UNIX network daemon management:** The Service Configurator pattern has been used in "superservers" that manage UNIX network daemons. Two widely available network daemon management frameworks are `inetd` [15] and `listen` [16]. Both frameworks consult configuration files that specify (1) service names (such as the standard Internet services `ftp`, `telnet`, `daytime`, and `echo`), (2) port numbers to listen on for clients to connect with these services, and (3) an executable file to invoke and perform the service when a client connects. These frameworks contain a master

Acceptor [8] process that reactively monitors the set of ports associated with the services. When a client connection occurs on a monitored port, the Acceptor process accepts the connection and demultiplexes the request to the appropriate pre-registered service handler. This handler performs the service (either reactively or in an active object) and returns any results to the client.

• **The Windows NT Service Control Manager (SCM):** Unlike `inetd` and `listen`, the Windows NT Service Control Manager (SCM) [12] is not a port monitor. That is, it does not provide built-in support for listening to a set of I/O ports and dispatching server processes "on-demand" when client requests arrive. Instead, it provides an RPC-based interface that allows a master SCM process to automatically initiate and control (*i.e.*, pause, resume, terminate, etc.) administrator-installed services (such as `remote registry access`). These services would otherwise run as separate threads within a single-service or a multi-service daemon process. Each installed service is individually responsible for configuring itself and monitoring any communication endpoints (which may be more general than I/O ports, *e.g.*, named pipes or shared memory).

• **The ADAPTIVE Communication Environment (ACE) framework:** The ACE framework [17] provides a set of C++ mechanisms for configuring and controlling services dynamically. The ACE `Service Configurator` extends the mechanisms provided by `inetd`, `listen`, and SCM to automatically support dynamic linking and unlinking of services. The mechanisms provided by ACE were influenced by the interfaces used to configure and control device drivers in modern operating systems. Rather than targeting kernel-level device drivers, however, the ACE `Service Configurator` framework focuses on dynamic configuration and control of application-level `Service` objects.

2.11 Related Patterns

The intent of the Service Configurator pattern is similar to the Configuration pattern [18]. The Configuration pattern decouples structural issues related to configuring services in distributed applications from the execution of the services themselves. The Configuration pattern has been used in frameworks for configuring distributed systems (such as Regis [19] and Polylith [20]) to support the construction of a distributed system from a set of components. In a similar way, the Service Configurator pattern decouples service initialization from service processing. The primary difference is that the Configuration pattern focuses more on the active composition of a chain of related services, whereas the Service Configurator pattern focuses on the dynamic initialization of service handlers at a particular endpoint. In addition, the Service Configurator pattern also focuses on decoupling service behavior from the service's concurrency strategies.

The Manager Pattern [21] manages a collection of objects by assuming responsibility for creating and deleting these objects. In addition, it provides an interface to allow clients access to the objects it manages. The Service Configurator pattern can use the Manager pattern to create and delete Services as needed, as well as to maintain a repository of the Services it creates using the Manager Pattern . However, the functionality of dynamically configuring, initializing, suspending, resuming, and terminating a Service created using the Manager Pattern must be added to fully implement the Service Configurator Pattern.

A Service Configurator often makes use of the Reactor [7] pattern to perform event demultiplexing and dispatching on behalf of configured services. Likewise, dynamically configured services that run for a long periods of time often execute using the Active Object pattern [22].

Administrative interfaces (such as configuration files or GUIs) to a Service Configurator-based system provide a Facade [1]. This Facade simplifies the management and control of applications that are executing within the Service Configurator.

The virtual methods provided by the `Service` base class are callback "hooks" [23] or "hook methods" [9]. These hooks are used by the Service Configurator to initiate, suspend, resume, and terminate services.

A `Service` (such as the `Clerk` class) may be created using a Factory Method [1]. This allows an application to decide the type of `Service` subclass to create.

3 Concluding Remarks

This paper describes the Service Configurator pattern and illustrates how it decouples the implementation of services from their configuration. This decoupling increases the flexibility and extensibility of services. In particular, service implementations can be developed and evolved over time independently of many issues related to service configuration.

The Service Configurator pattern also centralizes the administration of services it configures. This centralization can simplify programming effort by automating common service initialization tasks (such as opening and closing files, acquiring and releasing locks, etc). In addition, centralized administration can provide greater control over the lifecycle of services.

The Service Configurator pattern has been applied widely in many contexts. This paper used Java applets to demonstrate the application of the Service Configurator pattern in the Java run-time system. The ability to decouple the development of Java applets from their configuration into the Java run-time system exemplifies the flexibility offered by the Service Configurator pattern. This decoupling allows different applets to be developed in accordance with different service implementations. The decision to configure a particular applet into the Java run-time system becomes a run-time decision, which yields greater flexibility.

The Service Configurator pattern is also widely used in other contexts such as device drivers in Solaris and Windows NT, Internet superservers like inetd, the Windows NT

Service Control Manager, and the ACE framework. In each case, the Service Configurator pattern decouples the implementation of a service from the configuration of the service. This decoupling supports both extensibility and flexibility of applications.

4 Availability

The ADAPTIVE Communication Environment (ACE) provides an implementation of the Service Configurator pattern. ACE is freely available via the WWW at `www.cs.wustl.edu/~schmidt/ACE.html`.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[2] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[3] R. Gusella and S. Zatti, "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD," *IEEE Transactions on Software Engineering*, vol. 15, pp. 847–853, July 1989.

[4] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing*, vol. 3, pp. 146–158, 1989.

[5] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[6] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[7] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[8] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1996.

[9] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.

[10] D. Lea and J. Marlowe, "PSL: Protocols and Pragmatics for Open Systems," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[11] R. Gingell, M. Lee, X. Dang, and M. Weeks, "Shared Libraries in SunOS," in *Proceedings of the Summer 1987 USENIX Technical Conference*, (Phoenix, Arizona), 1987.

[12] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[13] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

[14] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[15] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[16] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[17] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[18] S. Crane, J. Magee, and N. Pryce, "Design Patterns for Binding in Distributed Systems," in *The OOPSLA '95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*, (Austin, TX), ACM, Oct. 1995.

[19] J. Magee, N. Dulay, and J. Kramer, "A Constructive Development Environment for Parallel and Distributed Programs," in *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 1–14, IEEE, Mar. 1994.

[20] J. M. Purtilo, "The Polylith Software Toolbus," *ACM Transactions on Programming Languages and Systems*, 1994.

[21] P. Sommerland and F. Buschmann, "The Manager Design Pattern," in *Proceedings of the 3rd Pattern Languages of Programming Conference*, September 1996.

[22] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

[23] S. Berczuk, "A Pattern for Separating Assembly and Processing," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

Using the Strategy Design Pattern to Compose Reliable Distributed Protocols*

Benoît Garbinato Rachid Guerraoui

Laboratoire de Systèmes d'Exploitation
Département d'Informatique
École Polytechnique Fédérale de Lausanne, Suisse
e-mail: bast@lse.epfl.ch

Abstract

Reliable distributed systems involve many complex protocols. In this context, *protocol composition* is a central concept, because it allows the reuse of robust protocol implementations. In this paper, we describe how the *Strategy* pattern has been recursively used to support protocol composition in the BAST framework. We also discuss design alternatives that have been applied in other existing frameworks.

1 Introduction

This paper presents how the Strategy pattern has been used to build BAST[1], an extensible object-oriented framework for programming reliable distributed systems. Protocol composition plays a central role in BAST and relies on the notion of protocol class. In this paper, we focus on the recursive use of the Strategy pattern to overcome the limitations of inheritance, when trying to flexibly compose protocols. In a companion paper [6], we have presented how generic agreement protocol classes can be customized to solve atomic commitment [10] and total order multicast [20], which are central problems in transactional systems and to group-oriented systems respectively. In [7], we also show how BAST allows distributed applications to be made fault-tolerant, by application programmers who are not necessarily skilled in reliability issues.

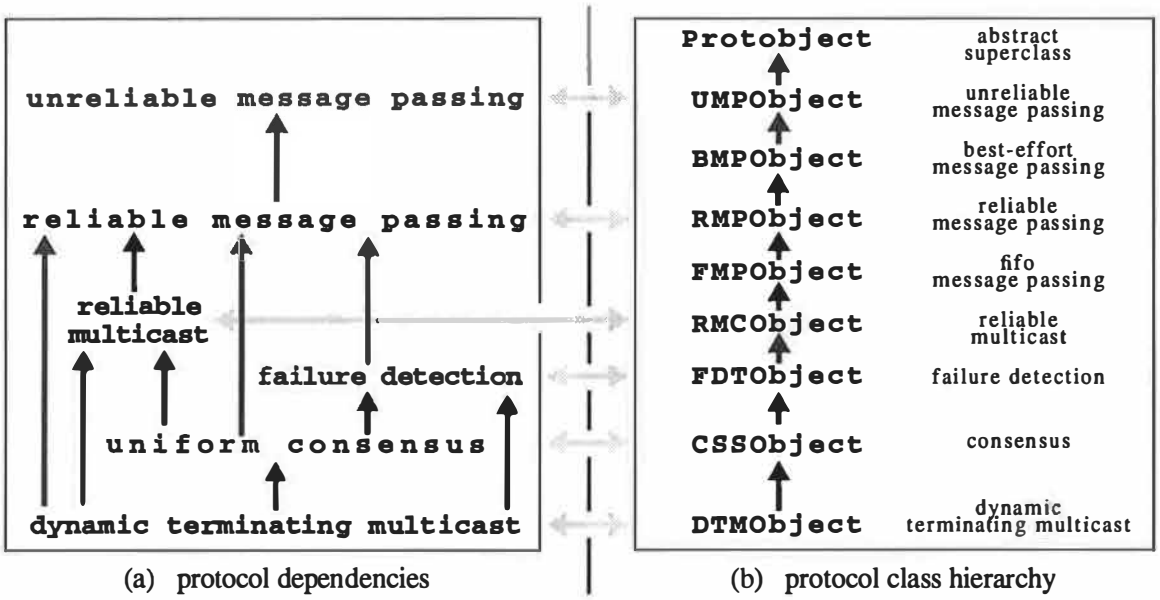
The BAST Framework

Building reliable distributed systems is a challenging task, as one has to deal with many complex issues, e.g., reliable communications, failure detections[2], distributed consensus, replication management, transactions management, etc. Each of these issues corresponds to some distributed protocol and there are many. In such a protocol "jungle", programmers have to choose the right protocol for the right need. Besides, when more than one protocol is necessary, the problem of their interactions arises, which further complicates programmers' task. The BAST framework aims at structuring reliable distributed systems by allowing complex distributed protocols to be composed in a flexible manner. For example, by adequately composing reliable multicast protocols with transactional protocols, BAST makes it possible to transparently support transactions on groups of replicated objects. It relies heavily on the *Strategy* pattern, which is recursively used to get around the limitations of inheritance as far as protocol composition goes. Our first prototype is written in Smalltalk [8] and is fully operational. It is currently being

*Partially supported by OFES under contract number 95.0830, as part of the ESPRIT BROADCAST-WG (number 22455)

[1]We named BAST after the cat-goddess of the Egyptian mythology: cats are known to survive several "crashes".

[2]A failure detector is a high-level abstraction that hides the timeouts commonly used in distributed systems [2].



(a) protocol dependencies (b) protocol class hierarchy

Figure 1: Protocols and Protocol Classes in BAST

used for teaching reliable distributed systems and for prototyping new fault-tolerant distributed protocols. Adding more and more protocol classes will help us to further test our approach. BAST has also been recently ported to the Java [9] programming environment. Performance is not yet good enough for practical application development, but we are currently working on performance evaluations and code optimization [7].

Overview of the Paper

Section 2 introduces the concept of protocol object as defined in BAST, and how it helps to structure distributed systems and to deal with failures. Section 3 discusses why inheritance alone is limited in supporting flexible protocol composition and presents how we applied the Strategy pattern to break these limitations. We also show how the Strategy pattern is transparently used in a recursive manner, and we present what steps have to be performed in order to extend BAST through protocol composition. Section 4 discusses various design alternatives, and compares our approach with other research works described in the literature. Finally, Section 5 summarizes the contribution of this paper, as well as some future developments in the BAST framework.

2 Protocol Objects

The BAST framework was designed to help programmers in building reliable distributed systems, and is based on protocols as basic structuring components. With BAST, a distributed system is composed of *protocol objects* that have the ability to remotely designate each other and to participate in various protocols. A *distributed protocol* π is a set of interactions between protocol objects that aim at solving *distributed problem* π. We use a ``$\pi$Object`` to name a protocol object capable of participating in protocol π, and we say that `$\pi$Object` is its *protocol class*. Each `$\pi$Object` provides a set of operations that implement interface protocol π, i.e., these operations act as entry points to the protocol. Abstract class `Protobject` is the root of the protocol class hierarchy.

With such broad definitions, any interaction between objects located on distinct network nodes is a distributed protocol, even a mere point-to-point

communication. For example, class `RMPObject` implements a reliable point-to-point communication protocol and provides operations `rSend()` and `rDeliver()` that enable to reliable sending and receiving, respectively, of any object[3]; callback operation `rDeliver()` is redefinable and is said to be triggered by the protocol. Note that such a homogeneous view of what distributed protocols are does not contradict the fact that some protocols are more basic than others. Communication protocols, for example, are fundamental to almost any other distributed protocol.

Dealing with Failures. Because failures are part of the real world, there is the need for *reliable* distributed protocols, e.g., consensus, atomic commitment, total order multicast. Reliable distributed protocols are challenging to implement because they imply complex relationships with other underlying protocols. For example, both the atomic commitment and the total order multicast rely on consensus, while the latter is itself based on failure detections, on reliable point-to-point communications, and on reliable multicasts. In turn, reliable multicasts can be built on top of reliable point-to-point communications. Figure 1 (a) presents an overview of some distributed protocol dependencies.

In BAST, protocol classes are organized into a single inheritance hierarchy which follows protocol dependencies, as pictured in Figure 1 (b). Each protocol class implements only one protocol, but instances of some `πObject` class can execute any protocol inherited from `πObject`'s superclasses. Protocol objects are able to run several executions of identical and/or distinct protocols concurrently.

[3]We mean here any object that is *not* a protocol object. Allowing the sending of protocol objects across the network implies the solving of the distributed object migration problem. We did not address this issue in our framework yet.

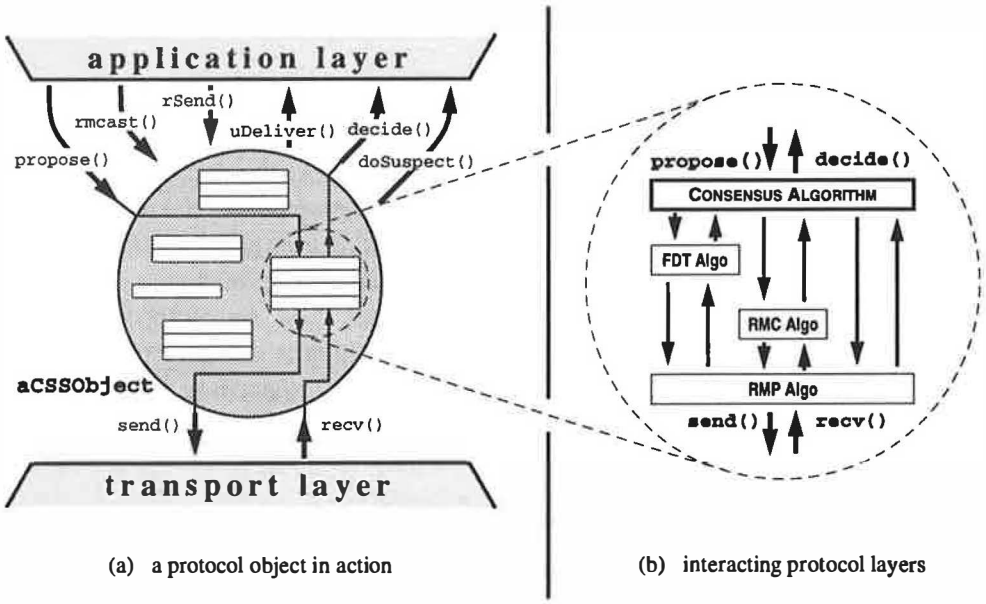
3 Strategy Pattern in BAST

Composing Protocols

With protocol objects, managing protocol dependencies is not only possible during the design and implementation phases (between protocol classes), but also at runtime (between protocol objects). This is partly due to the fact that protocol objects can execute more than one protocol at a time. In this context, trying to compose protocols comes down to answering the question "*How are protocol layers assembled and how do they cooperate?*".

Figure 2 (a) presents a runtime snapshot of `aCSSObject`, some protocol object of class `CSSObject` that implements an algorithm for solving the distributed consensus problem. The consensus problem is defined on some set σ of distributed objects as follows: all correct objects in σ propose an initial value and must reach agreement on one of the proposed values (the decision) [3]. Class `CSSObject` defines operations `propose()` and `decide()`, which mark the beginning and the termination of the protocol respectively [2]. Besides consensus, protocol object `aCSSObject` is also capable of executing any protocol inherited by its class, e.g., reliable point-to-point communications and reliable multicasts, as well as failure detections. In Figure 2 (a), `aCSSObject` is concurrently managing five different protocol stacks for the application layer, and issuing low-level calls to the transport layer. Focusing on the consensus stack, protocol composition means here to assemble various layers, each being necessary to execute the consensus protocol, into the protocol stack pictured in Figure 2 (b). The assembling occurs at runtime and creates a new stack each time the application invokes operation `propose()`.

Inadequacy of Inheritance Alone. With BAST, distributed applications are structured according to their needs in protocols: they are made of protocol objects, which act as distributed entities capable of executing various protocols. With this approach, it all comes down to choosing the right class for the right problem. We believe that inheritance is an



(a) a protocol object in action

(b) interacting protocol layers

Figure 2: Protocol Layers and Protocol Objects

appropriate tool to achieve this: by passing appropriate arguments to protocol operations and by implementing callback operations, programmers have the ability to tailor generic protocol classes to their needs. However, we claim that inheritance alone is not sufficient as far as protocol composition goes, because it does not offer enough flexibility. For example, inheritance does not allow for the easy implementation of a new algorithm for some existing protocol, and then to use it in various protocol classes that are scattered in the class hierarchy. Furthermore, inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*. These limitations lead us to seek an alternative solution for flexible protocol composition.

Protocol Algorithms as Strategies

According to Gamma et al., the intent of the *Strategy* pattern is to "*define a family of algorithms, encapsulate each one, and make them interchangeable*" [5, page 315]. This is usually achieved by objectifying the algorithm [4], i.e., by encapsulating it into a so-called *strategy* object; the latter is then used by a so-called *context* object. Making each π`Object` protocol class independent of the algorithm supporting protocol π is precisely what we need to be able to compose reliable distributed protocols in a flexible manner.

In the BAST framework, strategy objects represent protocol algorithms and they are instances of subclasses of class `ProtoAlgo`. A `ProtoAlgo` subclass that implements an algorithm for solving problem π is referred to as class π`Algo`. In the Strategy pattern terminology, a protocol algorithm, instance of some π`Algo` class, is a *strategy*, and a protocol object, instance of some π`Object` class, is a *context*. A strategy and its context are strongly coupled and the application layer only deals with instances of π`Object` classes, i.e., it knows nothing about strategies.

Strategy/Context Interactions. Figure 3 (a) sketches the way protocol objects and algorithm objects interact. On the left side, protocol object aπ`Object` offers the services it inherits from its superclasses, as well as the new services that are specific to protocol π. The actual algorithm implementing protocol π is not part of aπ`Object`'s code; instead, the latter uses services provided by strategy aπ`Algo` (on the right side of Figure 3 (a)).

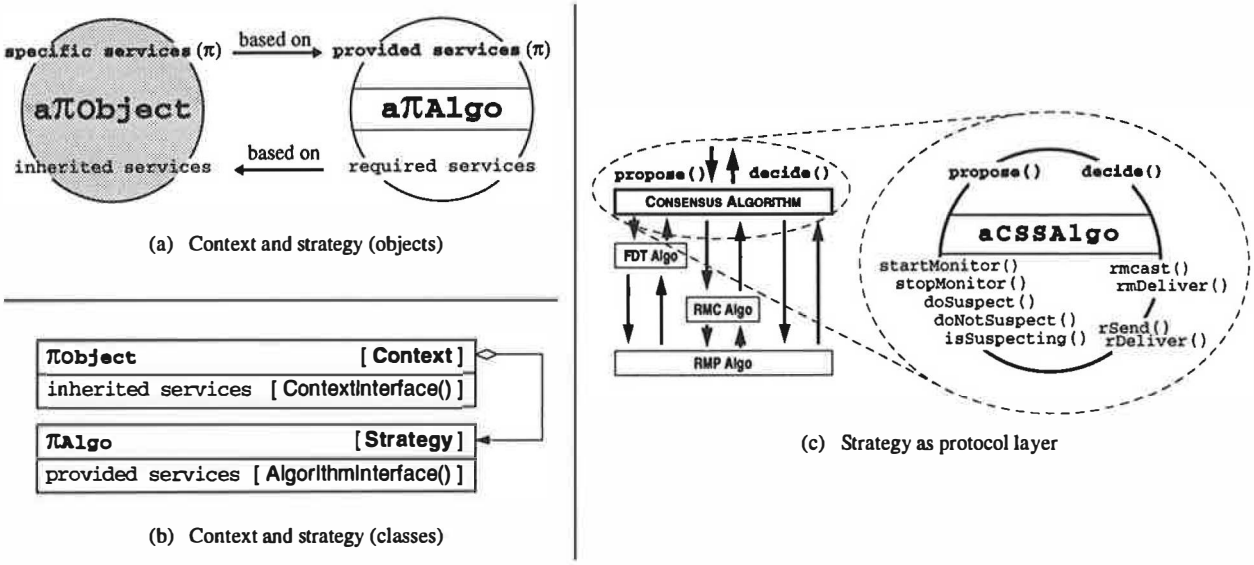


Figure 3: Strategy Pattern in BAST

Whenever an operation related to protocol π is invoked on `aπObject`, the execution of the protocol is delegated to strategy `aπAlgo`. In turn, the services required by the strategy to run protocol π are based on the inherited services of context `aπObject`. Such required services merely identify entry point operations to underlying protocols needed to solve problem π.

Each instance of class `πAlgo` represents one execution of protocol π implemented by that class, and holds a reference to the context object for which it is running; any call to the services required by the strategy will be issued to its context object. There might be more than one instance of the same `ProtoAlgo`'s subclass used simultaneously by `aπObject`. At runtime, the latter maintains a table of all strategies that are currently in execution for it. Each message is tagged to enable `aπObject` to identify in which execution of what protocol that message is involved, and to dispatch it to the right strategy. Figure 3 (b) presents the relationship between classes `πObject` and `πAlgo`, using a class diagram based on the Object Modeling Technique notation [19]. The correspondence between `πAlgo` strategy objects and layered protocol stacks is pictured in Figure 3 (c): at runtime, each strategy object represents a layer in one of the protocol stacks currently in execution.

Consequences. The context/strategy separation enables the limitations of inheritance to be overcome, as far as protocol composition goes. One could for example optimize the reliable multicast algorithm and use it in some protocol classes, while leaving it unchanged in others. Protocol algorithms could even be dynamically edited and/or chosen, according to criteria computed at runtime; this feature is analogous to the dynamic interpositioning of objects. There is a minor compatibility constraint among different protocol algorithms in order to make them interchangeable: new algorithm class $\pi\texttt{Algo}_n$ can replace default `πAlgo` in protocol class `πObject` *if and only if* $\pi\texttt{Algo}_n$ requires a subset of the services featured by `πObject`.

This approach also helps protocol programmers to clearly specify, for each protocol π, its dependencies with other protocols. One drawback of the Strategy pattern is the overhead due to local interactions between strategies and contexts. In distributed systems however, this overhead is small compared to communication delays, especially when failures and/or complex protocols are involved. More specifically, the time for a local Smalltalk invocation is normally under 100 μs, whereas a reliable multicast communication usually takes more than 100 ms when three or more protocol objects

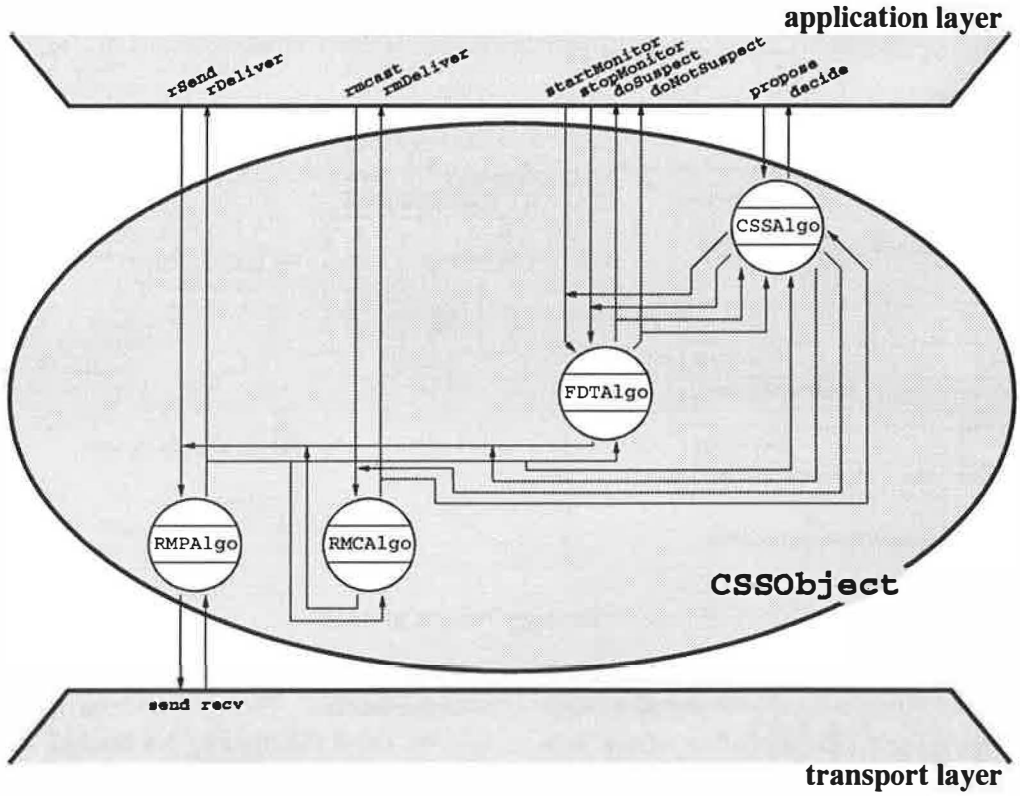


Figure 4: Recursive Use of the Strategy Pattern

are involved[4] (without even considering failures). The gain in flexibility clearly overtakes the local overhead caused by the use of the Strategy pattern.

Reliable Multicast: an Example

We now present how we implemented reliable *multicast* communications using the Strategy pattern. In BAST, class `RMCObject` provides primitives `rmcast()` and `rmDeliver()` that enable the sending and receiving, respectively, of a message `m` to a set of protocol objects referenced in `destSet`, in a way that enforces reliable multicast properties. The current implementation of class `RMCObject` relies on strategy class `RMCAlgo`.

Overview of the Protocol. The protocol starts when operation `rmcast()` is invoked on some initiator object $\texttt{aRMCObject}_i$, passing it a message `m` and a destination set `destSet`. In this operation, context $\texttt{aRMCObject}_i$ first creates a strategy $\texttt{aRMCAlgo}_i$, and then invokes operation `rmcast()` on it, with the arguments it just received. Strategy $\texttt{aRMCAlgo}_i$ builds message $\tilde{\texttt{m}}$, containing both `m` and `destSet`. It then issues a reliable point-to-point communication with each protocol object referenced in `destSet`; in order to do this, strategy $\texttt{aRMCAlgo}_i$ relies on inherited service `rSend()` of context $\texttt{aRMCObject}_i$. When message $\tilde{\texttt{m}}$ reaches $\texttt{aRMCObject}_t$, one of the target objects, operation `rDeliver()` is triggered by the protocol. Operation `rDeliver()` detects that $\tilde{\texttt{m}}$ is a multicast message and forwards it to $\texttt{aRMCAlgo}_t$, the strategy in charge of that particular execution of the reliable multicast protocol. When $\texttt{aRMCAlgo}_t$ receives $\tilde{\texttt{m}}$ for the first time, it re-issues a reliable point-to-point communication with each protocol object referenced in `destSet` (extracted from $\tilde{\texttt{m}}$), and then invokes `rmDeliver()` on its context $\texttt{aRMCObject}_t$, passing it message `m` (also extracted from $\tilde{\texttt{m}}$). This

[4] On a 10 Mbits Ethernet connecting Sun SPARCstations 20.

retransmission scheme is necessary because of the *agreement* property of the reliable multicast primitive, which requires that either all correct objects in `destSet` or none receive message `m` [2].

Recursive Use of the Strategy Pattern

When solving distributed problem π, one can strictly focus on the interaction between class `πObject` and class `πAlgo`, while forgetting about how other protocols are implemented. In particular, all protocols needed to support protocol π are transparently used through inherited services of class `πObject`. Those services might also be implemented applying the Strategy pattern, but this is transparently managed by inherited operations of `πObject`. In that sense, BAST uses the Strategy pattern in a powerful *recursive* manner.

The recursive use of the Strategy pattern is illustrated in Figure 4. The latter schematically presents a possible implementation of protocol class `CSSObject` presented in Section 3, which enables the solution of the distributed consensus problem by providing operations `propose()` and `decide()`. In Figure 4, the gray oval is context class `CSSObject`, while inner white circles are various `πAlgo` strategy classes (π being different protocols). Arrows show the connections between provided services (top) and required services (bottom) of each strategy class. Operations provided by class `CSSObject` are grouped on the application layer side (top). Each strategy class pictured in Figure 4 is managed by the corresponding context class in the protocol class hierarchy presented in Figure 1 (b).

Extending the BAST Framework

Basing the BAST framework on the Strategy pattern has the advantage of making it easily extensible. To illustrate this, we now present how we built `DTMObject`, a protocol class supporting the *Dynamic Terminating Multicast* (DTM) protocol [11] from existing contexts and strategies. The DTM protocol can be understood as a common denominator of many reliable distributed algorithms [12].

Overview of the Protocol. The protocol starts by the invocation of operation `dtmcast()` on an initiator object, passing it a message `m` and a set of protocol object references `destSet`. This invocation results in a reliable multicast of `m` to the set of participants objects. When message `m` reaches some participant, the protocol triggers operation `dtmReceive()`, passing `m` as argument. The participant object then computes a reply and returns it. Eventually, operation `dtmInterpret()` is triggered by the protocol on each non-faulty participant object, taking `replySet`, a subset of the participants' replies, as argument. The protocol insures that all correct participant objects get the same subset of replies, i.e., a consensus has been reached on that set.

Methodology for Extending BAST. A five steps methodology guides programmers in extending the BAST framework using the Strategy pattern. We illustrate each of these steps below, by presenting how the methodology was applied to the design of class `DTMObject`. Figure 5 summarizes the methodology.

1. Establish what services the new protocol class `DTMObject` provides, i.e., what operations are given to programmers wanting to use `DTMObject`; those operations are `dtmcast()`, `dtmReceive()` and `dtmInterpret()`.

2. Choose an algorithm implementing DTM and determine what services it requires, by decomposing it in a way that allows to reuse as many existing protocols as possible; those services are: consensus, failure detections, as well as reliable point-to-point and reliable multicast communications (see [11] for algorithmic details).

3. Implement the chosen algorithm in some `DTMAlgo` class; all calls to the above required services are issued to an instance variable representing the context object, i.e., an instance of class `DTMObject`.

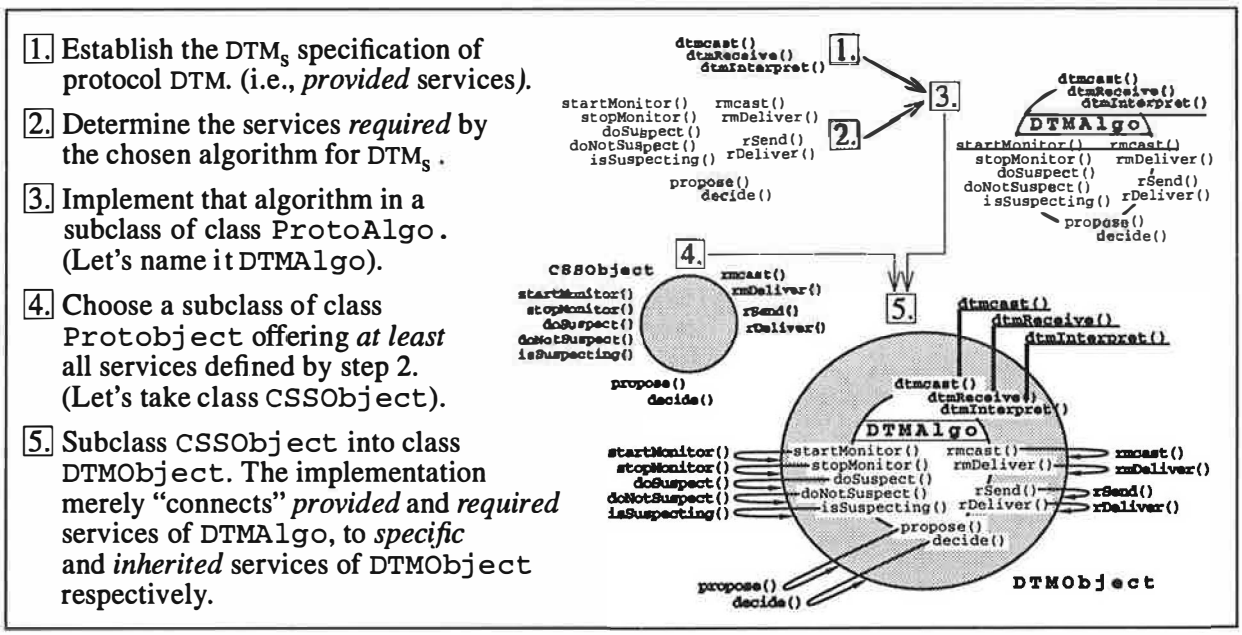


Figure 5: Extending BAST with Protocol Class DTMObject

4. Choose the protocol class that will be derived to obtain new class DTMObject; the choice of class CSSObject is directly inferred from step 2, since the chosen superclass has to provide *at least* all the services required by protocol DTM.

5. Implement class DTMObject by connecting services provided by class DTMAlgo to new DTM-specific services of class DTMObject, and by connecting services required by class DTMAlgo to corresponding inherited services of class DTMObject.

4 Design Alternatives

Our first implementation of BAST was not based on the Strategy pattern, i.e., distributed algorithms were not objects, and protocol objects were not capable of participating in more than one protocol execution concurrently. Furthermore, protocol composition was only possible through *single* inheritance[5].

[5]Remember that we used Smalltalk as implementation language for prototyping.

Because protocol objects are the basic addressable distributed entities in our approach, it is not possible to guarantee that there will never be more than one protocol execution involving each protocol object at a given time. For example, we cannot make sure that there will not be two concurrent multicast communications and/or transactions involving the same protocol objects. Allowing concurrency at this level is an essential feature. Moreover, as far as protocol composition is concerned, single inheritance is inadequate for offering a satisfactory degree of flexibility.

For all these reasons, we made BAST evolve through a second implementation of which the main goal was to overcome the limitations mentioned above. We now discuss some design alternatives that were considered in the process of implementing this second prototype of BAST, together with design issues that we studied from other existing frameworks described in the literature.

Multiple Inheritance and Mixins

Although our prototyping language does not offer multiple inheritance, assembling the various protocol layers through this code reuse mechanism is

very appealing[6]. The idea is to make each protocol class πObject implement only protocol π, while accessing all required underlying protocols through unimplemented operations; each protocol class is then an *abstract* class and we usually say it is a *mixin class* or simply a *mixin*. Before being able to actually instantiate a protocol object, one first has to build a new class deriving from all the necessary mixins.

There are three major drawbacks with this approach. First, protocol classes are not more ready-to-use components: a fairly complex multiple subclassing phase is now required. As consequence, programmers have to deal with protocol relationships "manually". Second, protocol layers can only be assembled through subclassing, and it is thus difficult if not impossible to compose protocol at runtime: in several programming languages, e.g., C++, classes are only compile-time entities. Third, we still have to manage concurrent protocol executions within the same protocol object, while this problem is handled nicely as soon as algorithms are manipulated as objects.

Toolbox Approach

Another possible approach to the reuse of protocol implementations is to provide programmers with a toolbox containing reusable components and associate them with design patterns. Both ASX [21] and CONDUITS+ [13] frameworks can be seen as such toolboxes. The ASX framework provides collaborating C++ components, also known as wrappers, that help in producing reusable communication infrastructures. These components are designed to perform common communication-related tasks, e.g., event demultiplexing, event handler dispatching, connection establishment, routing, etc. Several design patterns, such as the *Reactor* pattern and the *Acceptor* pattern, act as architectural blueprints that guide programmers in producing reusable and portable code. In CONDUITS+ [13], two kinds of objects are basically offered: *conduits* and *information chunks*, which can be assembled in order to create protocol layers and protocol stacks. Various patterns are also provided to help programmers in building protocols.

However, there is no such thing as protocol object in either of the above frameworks. Since our main intent is to provide programmers with a powerful unifying concept, the *protocol object*, we did not choose a toolbox approach for BAST. Furthermore, ASX does not promote protocol composition, whereas CONDUITS+ does it in a slightly different way than BAST, as we discuss below.

Black-box Framework

CONDUITS+ offers basic elements that helps programmers build protocol layers. The use of design patterns is motivated by the fact that traditional layered architectures do not allow code reuse across layers, which is precisely what CONDUITS+ aims at. Protocols can then be composed with CONDUIT+, at lower-level than BAST, through the assembling of conduits and information chunks, which are elementary blocks used to build protocol layers. In other words, the CONDUIT+ framework does not allow the manipulation of protocol layers as objects, but only the manipulation of *pieces* of protocol layers. Compared to BAST, protocol algorithms are further decomposed in CONDUIT+: conduits and information chunks are finer grain objects than BAST's strategies. Indeed, strategies represent protocol layers, while conduits and information chunks are internal components of protocol layers. CONDUIT+ goes one step further in the process of objectifying protocol algorithms.

This approach makes it easy for CONDUIT+ to be a pure *black-box* framework, while BAST combines features of both *black-box* and *white-box* frameworks[7]. With BAST, we are considering completely getting rid of inheritance but this issue has to be carefully studied, because it would have important consequences on the way BAST can be used by application programmers, i.e., those who have

[6] Ingalls and Borning have shown how reflective facilities of Smalltalk can be applied to extend the language with multiple inheritance [14], so we could have used that technique if we really wanted to.

[7] In a *black-box* framework, reusability is mainly achieved by assembling instances, whereas in a *white-box* framework, it is mainly achieved through inheritance. A black-box framework is easier to use, but harder to design.

very limited skills in fault-tolerant distributed algorithms.

Modeling Communications

Several systems model communications but do not really address reliability issues, e.g., STREAMS [18] and the *x*-Kernel [17]. AVOCA [24] defines the notion of protocol objects, but not in the sense that BAST does; furthermore, it mainly applies to high-performance communication subsystems. Other systems offer reliable distributed communications, either based on groups as elemental addressing facilities, e.g., CONSUL [15], ISIS [1] and HORUS [23], or based on transactions, e.g., ARJUNA [22].

Microprotocols and the *x*-Kernel

The work done by O'Malley and Peterson [16] is the closest to BAST that we could find. They extended the *x*-Kernel with the notion of microprotocol graph, and they described a methodology for organizing network software into a complex graph, where each microprotocol encapsulates a single function. In contrast, conventional ISO and TCP/IP protocol stacks have much simpler protocol graphs, with each layer encapsulating several related protocol functions. They argue that such a fine-grain decomposition allows for better tailoring of communication protocols to application needs; our conclusion concurs with theirs perfectly on that point. In their paper, O'Malley and Peterson mainly apply their approach to RPC communications (with only one very short discussion of what they call a *fault-tolerant multicast*). Compared to BAST, their approach is very close to what we have done and is based on the same basic assumption: composing (micro-)protocols is essential when it comes to customizing complex distributed applications (and fault-tolerance implies such complexity). In their terminology, what we call *problem* π is referred to as *metaprotocol* π.

There are also some important differences, however. They do not provide ready-to-use protocol classes to application programmers who are not skilled at understanding and/or building complex protocol graphs, whereas this is one of the main goals of BAST [7]. Moreover, their approach does not rely on design patterns. Similarly to CONDUIT+, they go one step further in their decomposition of protocol algorithms, by defining the notion of *virtual protocols*. The latter "*are not truly protocols in the traditional sense*" [16, page 131] : virtual protocols are actually used to remove IF-statements and to place them in the microprotocol graph instead. All those differences can be best understood by looking at the background domains of the BAST library and the *x*-Kernel respectively. The latter aims at helping system programmers to customize any communication protocol usually found in modern operating systems, while the former aims at providing ready-to-use protocol classes, in order to help any programmer to build fault-tolerant applications, and at allowing skilled programmers to build news fault-tolerant protocols easily.

Composing Protocol Stacks in HORUS

As far as protocol composition is concerned, the HORUS system enables the building of protocol stacks from existing layers only in a strictly vertical manner. Furthermore, it is based on groups as fundamental addressing *and* communication facility, and provides no framework and/or pattern for building *new* protocols layers. HORUS merely provides a finite set of ready-to-use protocol layers, which can only be composed around the group membership protocol.

With BAST, we have tried to model *any kind* of interaction between distributed objects, not only group communications. This is essential in order to deal with failures in an extensible way, because reliable protocols tend to be much more complex than normal communications. By making protocol objects BAST's basic distributed entities, we can build both the group model and the transaction model [6]. Furthermore, the Strategy pattern provides a powerful scheme for creating new protocols through composition.

5 Concluding Remarks

In this paper, we presented how protocol objects can help in building reliable distributed systems. We focused on how the Strategy pattern allows the limitations of inheritance to be overcome, when trying to compose protocols. As far as we know, BAST is the only environment to provide both a set of ready-to-use protocol objects for building fault-tolerant distributed applications, and a complete framework based on design patterns, for composing new protocols from existing ones. We see it as our contribution to the design of well-structured reliable distributed systems.

Our current prototype of BAST is fully operational and is available for Smalltalk and Java. At the moment, inheritance is still partly involved when composing distributed protocols; although a minor drawback, this does not make protocol composition as flexible as one might expect. This is due to the fact that programmers have to know something about the implementation of the protocol classes they reuse, namely their inheritance relationships. This is not surprising, since inheritance is known to violate encapsulation and to hinder modularity. Future work will consist of trying to decide if getting rid of inheritance, at least as far as protocol composition goes, is a good way to achieve even more flexibility. We are also extending BAST with new protocol classes, supporting frequently used protocols in reliable distributed systems, and optimizing existing protocol classes to improve performance. Further information about BAST can be found at *http://lsewww.epfl.ch/bast*; our public-free implementation is also available there.

References

[1] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.

[2] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 34(1):225–267, March 1996.

[3] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical report, Department of Computer Science, Yale University, June 1983.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming Proceedings (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1993.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In *European Conference on Object-Oriented Programming Proceedings (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, Linz (Autriche), July 1996. Springer Verlag.

[7] B. Garbinato and R. Guerraoui. Flexible protocol composition in BAST. Technical report, Operating System Laboratory (Computer Science Department) of the Swiss Federal Institute of Technology, March 1997.

[8] A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation.* Addison Wesley, 1983.

[9] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, Inc., October 1995.

[10] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In J.-M. Hélary and M. Raynal, editors, *Distributed Algorithms - 9th International Workshop on Distributed Algorithms (WDAG'95)*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100. Springer Verlag, September 1995.

[11] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: Bridging the gap. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 121–132. Springer Verlag, 1995.

[12] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177. IEEE Computer Society Press, June 1996.

[13] H. Hüni, R. Johnson, and R. Engel. A framework for network protocol software. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*. ACM Press, 1995. Special Issue of Sigplan Notices.

[14] D.H.H. Ingalls and A.H. Borning. Multiple inheritance in smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237. AAAI, 1982.

[15] S. Mishra, L. Peterson, and R. Schlichting. Experience with modularity in Consul. *Software Practice and Experience*, 23(10):1053–1075, October 1993.

[16] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[17] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abott. Rpc in the x–Kernel: Evaluating new design techniques. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 91–101, November 1989.

[18] D. Ritchie. A stream input-output system. *Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[20] A. Schiper and R. Guerraoui. Faul-tolerant total order "multicast" with an unreliable failure detector. Technical report, Operating System Laboratory (Computer Science Department) of the Swiss Federal Institute of Technology, November 1995.

[21] D.C. Schmidt. ASX: an object-oriented framework for developing distributed applications. In *Proceedings of the 6^{th} USENIX C++ Technical Conference*. USENIX Association, April 1994.

[22] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 1991.

[23] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'95)*, August 1995.

[24] M. Zitterbart, B. Stiller, and A. Tantawy. A Model for High-Performance Communication Subsystems. *IEEE Journal on Selected Areas in Communication*, 11(4):507–519, May 1993.

Supporting Synchronous Groupware with Peer Object-Groups

Jorge Paulo F. Simão*, José A. Legatheaux Martins,
Henrique João L. Domingos, and Nuno Manuel R. Preguiça*
Dept. of Computer Science,
Faculty of Sciences and Technology, New University of Lisbon
2825 Monte Caparica - Portugal
{jsimao, jalm, hj, nmp}@di.fct.unl.pt
*Work partially supported by PRAXIS XXI scholarships.

Abstract

We propose the peer object-group design pattern as a suitable architectural solution to structure and implement synchronous groupware applications. We discuss a reliable group-communication subsystem and a distributed objects model, implemented in Java, used to realize the approach.

1. Characterizing Groupware

"Computer Supported Cooperative Work" - CSCW, deals with the use of computer systems by people to cooperatively work on common tasks. Groupware is software specially built to allow people to work cooperatively.

Groupware and user interaction can be roughly classified in two broad classes - asynchronous or different-time, and synchronous or same-time. When using asynchronous groupware, users work not necessarily in the same time-frame and interact for long periods of time (e.g. in the joint development of a software project). When using synchronous groupware users work in a tightly-coupled manner during relatively short common time-frames (e.g. during a distributed meeting). The synchronous and asynchronous cooperation paradigms are not alternatives, but rather complementary; real work is most often performed alternating asynchronous work with synchronized periods.

We are in the process of investigating generic system-level services to support and allow simple development of robust groupware applications. In this text we will focus on system support for synchronous groupware. In particular, we will discuss structuring and programming abstractions based on group-communication and object-groups specially devised to help in the development of synchronous groupware applications (SGA).

A virtually common feature to all SGA is the provision of a shared workspace which users use to communicate and cooperate during a synchronous session. For acceptable productivity, users need to have an accurate notion of what the state of the shared workspace is. In particular, users should have mutually consistent views of the state of the workspace and should see each others actions as soon as possible. SGA are also interactive applications by nature, so it is required that the system responds and evolves accordingly to users expectations [1]. Users desire short or immediate response times; preferably, similar to that found in single user applications. Users do not find acceptable to wait a considerable amount of time to perform some operation (e.g. to update a shared object). Because users tend to divide/phase tasks into smaller sub-tasks and user communication and cooperation has multiple facets, SGA should be seen not as monolithic applications but rather as a collection of tools aggregated in the context of a single session (e.g. including a tool for shared drawing, a tool for message exchanging, a tool for text or document editing, tools providing audio and video channels, user activity awareness, coordination, etc.). From a software-engineering perspective, it is also preferable to use a generic multi-tool approach than to provide all the functionality from scratch in every application.

2. Design Alternatives to Support Distributed Synchronous Groupware

A commonly used architectural approach to support distributed SGA is the client-server paradigm. A central server is used to manage the shared workspace, to perform concurrency control on user accesses, and to provide other session related services (e.g. user activity awareness). User processes use the server to operate on shared resources and to disseminate information to other users. While this is a very well understood paradigm, and it is simple to realize,

it presents major drawbacks: fault-tolerance and scalability, since it is based on a central server. Moreover, performance can be somewhat injured by this architectural approach, although clients may replicate/cache parts of shared workspace in order to mitigate the problem. A variation is the centralized application-distributed interface approach, where a single application multiplexes user interaction and disseminates output across several user interfaces. It presents the same problems as the client-server approach, and is in general less flexible.

An alternative is the replicated-server, or object-group, approach. A group of servers actively replicates objects and/or service state. Even if a subset of servers crashes or becomes unreachable, the service will be available as long as some of them remain reachable (one or the majority - depending on consistency criteria). This approach is very suitable for many distributed fault-tolerant services, but still presents some drawbacks in the context of SGA. Because users want to have accurate views of the shared workspace, and because users actions are largely driven by other users actions, extra mechanisms for event notifications are required.

Preliminary experience on scalable, fault-tolerance, distributed systems has suggested that migrating complex system functionality from servers to clients may be a suitable design option. This argument, the need for flexibility in tool building, and the low-latency requirements of SGA, suggests, in our view, a much more natural approach - the peer object-group approach.

3. The Peer Object-Group Design Pattern

In the peer object-group approach the shared workspace managed by SGA is materialized as a collection of objects replicated amongst users local environments. Each local environment holds a replica for every object the associated user is currently accessing or working on. The set of replicas for a given object constitutes a (peer) object-group. Consistency amongst the replicas is kept by a group-communication subsystem implementing appropriate consistency criteria. Shared objects are mapped to object-groups and operations on the objects are mapped to (reliable) multicast operations. Figure 1 schematically illustrates the model.

Users gain access to objects by dynamically joining the corresponding object-groups - which may involve the transparent transfer of the object's current state to the local replica. When no longer interested in the objects, users leave the object-groups.

Users keep accurate views of the shared state since updates are received by all object-group members; no provisions for additional notification mechanisms is required. Latency in object manipulation is improved because no intermediate entities are present. Fault-tolerance and availability is also improved; a K-degree of fault-tolerance is achieved as long as K+1 members keep copies of shared objects.

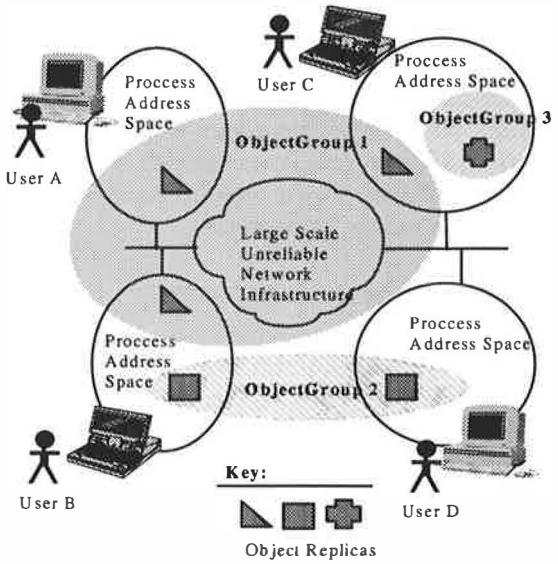


Figure 1 - The peer object-group design pattern.

Different shared-objects may have different replication consistency requirements, meaning that the underlying group-communication sub-system should provide group-protocols with different service semantics. The selection of object-groups granularity must inevitably be tool and protocol driven; the lighter-weighted the protocols are, the finer the granularity can be.

Object persistence is not addressed by this bare model. While it is desirable that some objects outlive sessions, that facility is provided by the asynchronous groupware support, and will not be discussed in this paper.

4. Object-Oriented Group-Protocol Implementation and Composition

The essential component to realize the peer object-group approach is the group-communication subsys-

tem, which provides group membership and message passing services. Those services are implemented by group-protocols and accessed through the combined use of a user-to-protocol service request interface and a protocol-to-user event notification interface. The former includes methods for message sending and multicasting as well as group management. The later includes methods for message delivery and group membership view change notifications.

Implementing group-protocols is a complex task, mainly because they must convert an unfriendly system environment into a friendly one. A good engineering option is to use a modular approach in designing and implementing group protocols. Multiple (micro-)protocol layers, each implementing some specific service, are stacked to built complex protocol services [2].

To realize the peer object-group we have implemented an object-oriented framework to allow the convenient implementation and composition of group-protocols. Because we want to maximize flexibility, allow application and system components to be loaded on-demand, and support heterogeneity, the Java language was a natural choice [3]. The integration with the Web was an additional motivation.

In our framework protocol layers are implemented as objects of special classes, which implement group services related programming interfaces. Complete protocol structures (or stacks) are built attaching protocols objects together. To allow simple construction of protocol structures, protocol structures description strings and generators are used. Description strings convey information about which protocols should be used to build a particular protocol structure and the topological relationships between the layers. Protocol structure generators parse strings and generate the correspondent protocol structures, by dynamically loading the layer classes and creating the layer objects.

In implementing specific group-protocols we have considered SGA specifics. Because users objects working-sets are expected to change often during the lifetime of a session and users should be able to enter and leave sessions dynamically, dynamic lightweight group membership services were used. In particular, we have specified a new membership and reliable multicast service semantics - *linear convergent synchrony*, which is weaker than the "standard" *view synchrony* [4], but can be implemented by protocols which incur in less overhead for group membership management. The semantics and implemented protocol are linear, in the sense that no view merging is allowed, because we assume that state reconciliation due to network partitions is performed using the external data storage services. The protocol uses a specially tailored FIFO reliable multicast protocol.

We have also experienced with optimistic ordering techniques to reduce system response-time. In particular, the Undo/Redo delivery paradigm was used to reduce update latency [5]. In this paradigm messages are delivered locally while asynchronously multicasted to the group. If ordering conflicts arise, some previously delivered messages/updates are undone. Object operations semantics (e.g. the commutative property), is explored to reduce the probability of conflicting updates. The protocol sits on top of a sequencer based total ordering and state transfer protocol, which achieves high levels of concurrency even during process joins.

5. Object-Groups Management and Session Services

In addition to a group-communication subsystem SGA programming can benefit from the provision of other more specific services. Mechanisms and services are required for the naming and binding to sessions, for the management of the object-groups in the shared workspace, and to enable user activity awareness.

We have defined and implemented an extensible distributed object model to structure and implement SGA, which tackle the above issues in an integrated manner. In addition to the collection of peer object-groups which constitutes the shared workspace, we have introduced the notion of fully replicated Session object. A Session object is supported by a special bootstrap object-group, which all user processes must join to enter a session. Binding information required to enter a session is fetched from an external binding service.

A Session object's main purpose is to store and manage directories which hold information about created object-groups and users participating in the session. Object-groups information includes binding and management data (e.g. protocol structures description and replication options). User information includes human readable data about human users (e.g. user full name, user photography, e-mail address, the

Web home-page, etc.). Both object-groups and users are identified by names, and are represented as Java classes which can be application derived to convey additional information. Conceptually, we abstract an application as a collection of shared object(-groups) and users organized around the fully replicated Session object. Figure 2 depicts an intuitive view of the distributed objects model.

From an application programmer perspective, she/he can invoke the methods of a Session object to create, destroy, join or leave object-groups and to obtain information about users. A reactive programming style can also be used to act on session related events (e.g. a user entering or leaving a session, or an object-group being created or destroyed).

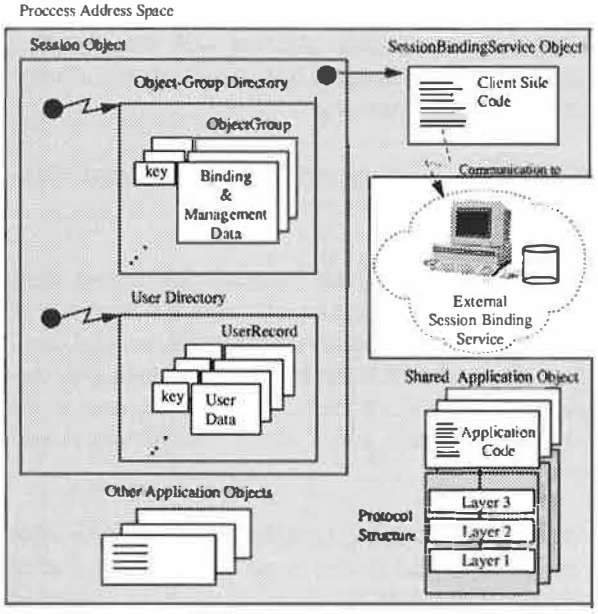


Figure 2 - Objects conceptual model.

6. Experience and Future Work

As described, we have developed up to this point the framework for protocol composition, a set of stackable group-protocols and the distributed objects model. We have tested the suitability of our ideas implementing a demo white-board tool. It is a simple tool which manages a shared drawing canvas and requires only one object-group to be implemented. It was tested with only a small number of users in a local network. In this restricted setting, system response has reveled to be quite acceptable, i.e. system performance did not suffer significant degradation when operating on replicated shared objects. Additional experience and performance measures are required to analyze system behavior in more general environments.

Many potential work directions were revealed during the course of our work. We plan to continue the process of specifying suitable group-communication semantics and implementing new protocols. In particular, we expect to develop layers for light-weighted groups - multiplexing many groups into a small number of groups, in order to increase system performance when many fine granularity object-groups are used. This may call for the definition of multiple-group service semantics. The issue of failure-detectors consistency will also be addressed, i.e. ensure that the membership layers of several protocol stacks have similar views of what the connectivity state is. Also, we expect to tackle the always important issue of security and access control.

We also plan to develop additional tools and applications, to help validating more clearly the usefulness of the abstractions discussed in this paper. We will consider enhancing our object model with additional structuring abstractions and common services as more experience is gained. Finally, we intend to build a stub-compiler to simplify the task of shared objects programming.

References

[1] C.A. Ellis, S.J. Gibbs, and G.L. Rein, *Groupware - Some issues and experience*, Communication of the ACM, vol. 34, n.1, Jan. 1991.

[2] van Renesse, Robbert, Birman, Kenneth P., Friedman, Roy, Hayden, Mark, and Karr, David A., *A Framework for Protocol Composition in Horus*, in proceedings of the 14th IEEE International Conference on Distributed Computing Systems, 1994.

[3] Gosling, James, McGilton, Henry, *The Java(tm) Language Environment: A White Paper*, Sun Microsystems, 1995.

[4] Kenneth P. Birman, and Thomas A. Joseph, *Exploiting Virtual Synchrony in Distributed Systems*, Department of Computer Science, Cornell University, 1987.

[5] Karsenty, Alain, and Beaudouin-Lafon, Michel, *An Algorithm for Distributed Groupware Applications*, in proceedings of the 13th IEEE International Conference on Distributed Computing Systems, 1993.

Reliability with CORBA Event Channels

Xavier Défago Pascal Felber Benoît Garbinato Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation
Département d'Informatique
École Polytechnique Fédérale de Lausanne, Switzerland
e-mail: {xavier,pascal,benoit,rachid}@lse.epfl.ch

1 Overview

Several application domains such as finance, process control, and telecommunications, have strong reliability requirements. Typically, such applications tend to avoid having a single point of failure, and need to communicate with reliable primitives that prevent message loss and ensure atomicity guarantees. Among such applications, we have focused on reliable *notification-based* applications, such as trading systems and news agencies, where *producers* need to *reliably* deliver information to a set of *consumers*. Developing such applications is greatly eased with a middleware providing *reliable broadcast* semantics [7].

Group-oriented systems like Isis [3], Horus [9], Totem [2] or Transis [1], provide reliable broadcast primitives and are generally considered to be good candidates for implementing reliable notification-based applications. Nevertheless, these systems are proprietary solutions with limited portability and interoperability. Although efforts have been made recently to achieve better modularity (e.g., in Horus), the infrastructure of group-oriented middlewares usually consists of several layers that are not necessarily required at upper levels and usually turn out to be performance penalizing.

Orbix+Isis [8] is an effort at supporting replication of CORBA objects transparently, by integrating Isis in Orbix. This approach requires a modification of the ORB and leads to a non-standard and non-interoperable solution. The Object Group Service [6] provides replication of CORBA objects without using heavy-weight group communication toolkits (e.g. Isis) and would provide the degree of reliability required by our application class. The tradeoff is performance degradation since it introduces replicated intermediary objects.

We present here a way to augment CORBA with a reliable broadcast facility. Our approach is pragmatic in the sense that it requires no modification of the Object Request Broker, and we do not build a new CORBA service from scratch. Instead, we add reliability features to the existing CORBA Event Service, which already provides multicast-like communication. The extension we introduce requires no modification of the CORBA specification, and can be applied to any standard Event Service implementation, without any communication overhead. The resulting service, called *Reliable Event Service*, adequately fits the required semantics of reliable notification-based applications. It constitutes an interesting light-weight and open alternative to existing group-oriented systems.

2 Reliability Issues

We consider notification-based applications where communication is decoupled between consumers and suppliers of information, with specific reliability requirements. This type of applications is widespread in domains like process control, finance, or telecommunications.

The use of CORBA for such applications bears many advantages over other approaches. The portability and interoperability aspects of CORBA are strong assets. The paradigm offered by the event channels is well adapted to notification-based ap-

plications since it provides a flexible model for asynchronous communication among distributed objects. Furthermore, relying on one-to-one communication to implement this functionality would require some amount of bookkeeping to keep track of the consumers. Finally, depending on the implementation, there is a potential for the Event Service to be scalable while it is clearly not the case with one-to-one communication primitives.

2.1 Limitations of the Event Service

Since the Event Service is based on a centralized architecture, where a channel is just another CORBA object, it introduces a single point of failure. Furthermore, the CORBA specification is vague concerning the quality of service provided by event channels. It states that the Event Service does not need to provide stronger semantics than "best-effort" delivery of the events, although implementors of the Event Service are advised to provide various semantic levels for their channels. The problem of the centralized architecture of the event channels may in two ways:

- *Replicate the event channels.* Event channels are replicated, and hence, are no longer a single point of failure. This approach, used in Isis News [3], requires the use of specific protocols, like group communication [4], to keep replicated objects consistent.

- *Decentralized architecture.* In a decentralized architecture an event channel is not implemented as a single physical object but rather as a collection of collaborating objects. This approach makes it possible to build a protocol based on IP-multicast rather than point-to-point communication, thus improving efficiency and scalability.

A solution to the lack of clearly specified semantics requires the definition of a standard quality of service to be expected from any implementation of the Event Service and a standard way to select it. The specification may define different levels of quality of service, from which the application programmer may choose. Currently, a valid implementation of the Event Service needs to be at least "best-effort". In other words, it puts no actual requirement on the delivery semantics since "best-effort" is a subjective description rather than a real property. Since a vague and minimal description is not suitable for reliable applications, we describe a protocol in Section 3 that extends any Event Service to make it reliable.

3 A Reliable Event Service

We introduce here the Reliable Event Service that provides *reliable* event channels, by extending the quality of service of any existing (unreliable) Event Service. The approach we adopted provides the exact quality of service required by the application class considered, and focuses on providing good performances. Furthermore, it is orthogonal to the architecture (centralized/decentralized/replicated) of the Event Service that it extends.

The semantics we associate with the Reliable Event Service are close to those of a Reliable Multicast primitive [7]. This primitive ensures that different clients receive the same set of messages. An informal definition of this primitive could be the following: if a correct object multicasts a message m, then all correct objects eventually deliver m. Furthermore, if a correct object delivers a message m, then m was previously multicast by some object and all other correct objects will eventually deliver m. Briefly, Reliable Multicast has two properties: *at-most-once* and *atomicity* (*all-or-nothing*).

Ideally, we would use a reliable multicast primitive, but its strong properties have a very high cost in terms of communications overhead. In a typical implementation of this primitive, the number of messages generated belongs to $O(n^2)$ and it requires that each consumer keeps a list of all the other clients. In the context of a diffusion network (e.g., Ethernet) where the complexity of a multicast is $O(1)$, the complexity of the reliable multicast is still $O(n)$. Since the cost increases proportionally with the number of destinations, it is not scalable.

Our mechanism has weaker properties than a Reliable Multicast, but it suits our requirements for

reliability and does not change the complexity of the underlying communication. It is split into three parts. The first part consists in detecting when a message has been lost, in order to emit a notification. The second part helps to reduce the probability of actual loss by retrying unsuccessful transmissions. Finally, the last part ensures that messages are delivered in a FIFO manner.

3.1 Notification of Message Loss

Since there is no time bound on the delivery of messages, it is not possible to distinguish a lost message from a slow one. Hence, we consider the message to be lost in both cases.

To detect when a message is lost by the channel, we add some extra information to each message: a unique message identifier. Each producer has a unique identity given by its CORBA object reference. This tag makes messages issued by two different producers distinguishable. In order to differentiate messages issued by the same producer, we add a second field holding a local identifier (id). This id consists of a sequence number that is incremented each time a new message is sent. Therefore, clients will eventually detect lost messages based on missing sequence numbers. If the underlying event channels are not FIFO, the client may assume that a message is lost while it is only delayed. In that case, the client will launch the replay protocol (see below), and discard duplicate messages.

3.2 Message Replay

When a client detects the loss of a message, it contacts the producer by using the CORBA reference embedded in the message identifier. The client issues a request for the lost message using a synchronous remote method invocation and waits for a reply (see Figure 1). If the producer has not crashed in the meantime, the message will be resent and the client may continue. If a problem occurs (e.g., the producer has crashed) the reply is an exception and the client is supposed to react adequately. This approach is actually based on the principle of negative acknowledgments. In order to be able to resend a message, the producer needs to keep a buffer with every message it sends.

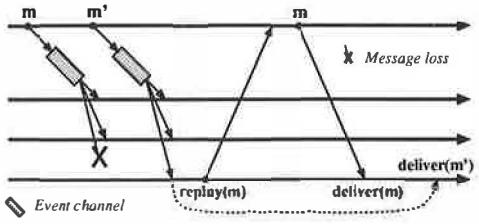


Figure 1: Replay of a lost message.

When the loss of a message is not recoverable, the most sensible approach consists in issuing an exception to the handled by the application. Since the adequate reaction to such a loss varies from one application to another, we leave the responsibility of reacting properly to the application programmer.

3.3 Ensuring FIFO Ordering

Since our protocol is aimed at working with any implementation of the event channels, we face an additional problem. If the underlying protocol ensures that received events are delivered in the same order than they were sent (FIFO property), replaying lost messages breaks this property. Hence, to avoid this problem, we add a mechanism that guarantees a FIFO delivery of events. This mechanism, illustrated in Figure 1, is an adaptation of the FIFO multicast presented in [7].

We first need to distinguish the reception of a message from its delivery. We call $receive(m)$ the reception of the message m by the lower protocol layer, and $deliver(m)$ the delivery of the message m from the lower layer to the upper layer. In some situations (e.g. upon a message loss) a message m', sent after m, may arrive before m. In other words, $receive(m')$ precedes $receive(m)$.

In order to ensure the FIFO property, the delivery of m' is delayed until m has been received and delivered. This implies that the FIFO order of delivery is preserved for the upper layer. In other words, $deliver(m)$ precedes $deliver(m')$. The FIFO property is thus guaranteed by our protocol, whether or not the underlying communication channel delivers the events in a FIFO order.

3.4 Appropriate Reaction

When a message has been lost and is no more available, the client has to react accordingly. The most appropriate reaction depends on the application. A non-exhaustive list of possible reactions to the loss of a message is:

- *Ignore (trivial case).* The lost message is ignored. There was no need for our protocol and reliability is not necessary.
- *Quit.* The client is considered faulty, and hence, decides to commit suicide.
- *Quit & Recover.* The client is considered crashed, but it subscribes again to the event channel, as if it were just starting to listen to the event channel. In the initialization phase, a producer may send initial information to the newcomer.
- *Warning.* A warning message is issued to the end-user, telling that some information might not be up-to-date.

To satisfy the needs of a large number of applications, the most sensible approach consists in issuing an exception whenever a message cannot be retransmitted. This leaves the responsibility of reacting properly to the application programmer.

In order to guarantee the atomicity of delivery, it is necessary for the client not to be considered correct when it fails to deliver a message. Therefore, the only reactions that guarantee atomicity are *"Quit"* and *"Quit & Recover"*.

4 Implementation Issues

When evaluating the relevance of using a middleware for the development of notification-based applications, one of the main concerns is to rely on a standard definition rather than features specific to a particular vendor. Since these applications are expected to evolve over a long period of time, portability is a strong requirement.

Our current implementation suffers from a number of limitations inherent to the underlying Event Service that we use, i.e. IONA's OrbixTalk. In particular, it supports only the push model defined in the Event Service specification, and does not allow to chain event channels (i.e., there must be at most one event channel between a consumer and a supplier). More information can be found in [5]

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication sub-system for high availability. In *Proceedings of the IEEE 22nd International Symposium on Fault Tolerant Computing Systems*, 1992.

[2] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[3] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual*. Dept of Computer Science, Cornell University, September 1990.

[4] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[5] X. Defago, P. Felber, and R. Guerraoui. Reliable corba event channels. Technical report, Département d'Informatique, EPFL, Switzerland, May 1997.

[6] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the IEEE 15th Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, October 1996.

[7] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[8] IONA and Isis. *An Introduction to Orbix+Isis.* IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[9] R. van Renesse, K. Birman, and R. Cooper. The HORUS system. Technical report, University of Cornell, NY, 1993.

Interactive-Group Object-Replication Fault Tolerance for CORBA*

Brent E. Modzelewski†
Electrical and Computer Engineering Dept., Worcester Polytechnic Institute
David Cyganski, Ph. D.
Electrical and Computer Engineering Dept., Worcester Polytechnic Institute
Marian V. Underwood
Lockheed Martin Corporation, Government Electronic Systems

Abstract

As more and more computers and workstations enter the workplace they are inevitably connected to a network. Networks provide the interconnection necessary for computers to share common data, peripherals and other system resources. Distributed computing allows network applications to access functions or processes on remote computers. Developing network applications specifically to interact and draw upon resources of multiple computers creates the groundwork for a distributed system or distributed computing environment (DCE).

An ideal distributed system is self monitoring and resilient to failures. In the event of a failure the system should dynamically reconfigure itself with automatic fail-over for applications that fall victim to the fault. Transparency of fault tolerant mechanisms is desirable, especially when introducing legacy applications into the distributed system. The reduction of application development efforts heavily relies on the availability of portable, non-invasive, fault tolerance providing extensions, which introduce mechanisms for uninterruptible service by insertion into existing distributed applications.

To test the potential for addressing some of these desired capabilities for a distributed system implemented within a CORBA distributed computing environment, the Interactive-Group Object-Replication (IGOR) system was developed. IGOR is a system of objects that provides fault tolerance through object replication by arranging replicas in fault tolerant groups which interact to provide access to redundant data and services. For purposes of portability, interoperability and to evaluate the CORBA environment, IGOR was designed with the constraint of lying entirely within the CORBA architecture and using IIOP as the communication protocol. This guarantees its portability over changes in platform and network technologies. The IGOR system is reconfigurable and its fault tolerance mechanisms are completely transparent to client applications.

1. Introduction

A plethora of computers and workstations enter the workplace each year and play an increasingly important role as a digital tool used by people worldwide. With the increase of the use of computers comes the increase of stored information and multi-client services. However experience has shown that such information sources and services quickly become decentralized, then isolated and as a result, not interchangeable. Isolation is due to the disintegration of applications segregated by disparate hardware and operating systems, which lack the interoperability and robustness necessary for seamless information and service sharing.

A distributed computing environment (DCE) implies an environment in which interoperation is not only possible, but is fundamentally inherent and natural. Distributed computing exploits the computational power of many computers, integrating the entire system into a single functional unit. Load balancing, parallel processing and distributed objects are major technologies that have evolved which aid in the implementation of full fledged distributed systems.

An ideal DCE provides the programmer with automated tools that permit construction of distributed applications (clients and servers). Distributed applications access functions on remote computers while giving the appearance of locally executed functions.

* This research supported by Lockheed Martin Corporation, Government Electronic Systems
† Corresponding author: brentm@ece.wpi.edu

Furthermore, the user need not be aware of the remote or local nature of the processes, nor the movement and conversion of the data involved.

Usually the preconceived notion of a distributed system is one in which a specific task is being carried out by a small set of networked computers running the same operating system. That may have been true several years ago, but today a distributed system can be much more diverse. Distributed systems can span many computers and interoperate with virtually any operating system or hardware platform available on today's market. The magnitude of distributed systems varies greatly depending on the intended purpose. A distributed system can contain a few interoperating computers, or may span the entire Internet.

2. Overview

A component or object based architecture can greatly benefit a distributed system. Construction of a system with objects vastly increases modularity; the interchangeable nature of individual system components translates into design and implementation flexibility for the applications engineer. Ideally, upgrades to the system can be done on a component basis, while the system remains up and running. Zero downtime is a very attractive feature of an object based distributed system, especially for mission critical applications. Providing uninterruptible system-wide service is a difficult and complex task. Many standards have been developed to aid in the composition of such systems.

One of the distributed systems standards that has been devised since the emergence of the object paradigm is called the Common Object Request Broker Architecture (CORBA) [2]. CORBA is one component of the Object Management Architecture (OMA) and was developed by the Object Management Group (OMG) [1]. CORBA is a well defined robust standard that is component (object) based, is supported on virtually all hardware platforms, and is fully interoperable and portable.

On the surface, the CORBA standard for distributed objects appears to be another type of Remote Procedure Call (RPC) [3] implementation. On closer inspection, CORBA proves to offer much more than just predefined procedure calls with static parameters. Execution of remote objects, parameter marshaling, multiplatform interoperation, interface portability, dynamic interface invocation, variable parameters and platform independent data types are some of the features of CORBA that do not exist in standard RPC, OSF/DCE or message passing standards, such as Message Passing Interface (MPI) [5] and the like.

While many aspects of CORBA are attractive for large scale distributed system development, it does not inherently support more than rudimentary levels of fault tolerance. To implement basic fault tolerance in CORBA without involving external mechanisms, server applications can be cloned and distributed throughout the network to provide high availability of services to clients. Cloning merely creates redundant copies of the server application, so services are more readily available to client applications. Unfortunately this does not take into account the mutable state of the application at hand. Since no data synchronization takes place between clones, this presents a low level of fault tolerance and is unacceptable for many fault critical applications.

During failures, certain CORBA Object Request Brokers (ORBs), such as Visigenic Software's VisiBroker [6], can automatically fail-over to an application that can provide a desired service. However there is no guarantee of appropriate object state. Fail-over occurs transparently to client applications, thus providing a layer of isolation between the client and the system's fault tolerant mechanisms.

Replicating server applications is another technique similar to cloning that provides high availability of services to clients. Object replication is meant to not only provide availability of services, but also to maintain strict data consistency between objects [4].

3. IGOR Architecture

In response to the need for a fault tolerant system and our desire to test CORBA with respect to its support for easily insertable object behavior extensions, we have developed the IGOR (Interactive-Group Object-Replication) system. Object replication was selected as the basis for implementation of IGOR's fault tolerance mechanism. IGOR is a system of interacting objects that provides mechanisms for the creation of a reconfigurable fault tolerant system, with the additional and strategically important constraint of lying entirely within the CORBA architecture. Layered on CORBA, IGOR yields a portable, interoperable and modular design, that will remain portable over

changes and improvements in the CORBA standard and changes of platform, operating system and communication technologies.

Much of our attention has been directed towards development of a fault tolerant system which reduces the invasiveness of the underlying fault tolerant mechanisms with respect to implementation and operation. Our aspiration for IGOR was to provide a tool that would ease the development of fault tolerant distributed applications, while simultaneously embracing the introduction of legacy (CORBA and non-CORBA) applications into the fault tolerant arena.

An object grouping scheme has been devised for the IGOR system to facilitate fault tolerance by redundancy (object replication). Distributed replica server applications enroll themselves into fault tolerant groups through an IGOR registration process. These groups are actually logical representations that associate like server applications that share a common data set. Each fault tolerant group functions separately as a single logical unit and group members interact with each other to maintain intragroup data consistency. Client applications benefit by the group's high availability of services and redundant data.

Fault tolerant groups in IGOR are resilient to partial failures and provide these fault tolerant services transparently to client applications. In fact, the client object does not need to be aware that the server application which it is accessing is a member of a fault tolerant group. The client code is identical whether the client object is connected to an IGOR fault tolerant object or a single non-fault tolerant object. Because of this flexibility, fault tolerance may be added to the system even after client applications have already been deployed. This is done by simply replacing the non-fault tolerant server objects with their IGOR fault tolerant counterparts. No code modification or recompilation of the client application is required for the addition of IGOR fault tolerance.

A single IGOR Registry Service acts as the governing body for fault tolerant group enrollment. The purpose of the registration process with the IGOR Registry is to ensure that fault tolerant groups consist of only objects of identical type. The IGOR Registry is responsible for recording the logical arrangement and association of all replica groups; of course this information is persistently stored concurrently in a redundant object database. To keep track of all the fault tolerant replica groups the Registry constructs a binary tree consisting of all groups, this binary tree being called the Group Tree. A single Group Tree represents active fault tolerant groups for the entire IGOR system. The purpose of the Group Tree is only to facilitate organizing replica groups in a meaningful fashion to ease the Registry's task of group management. To arrange group members, each node on the Group Tree contains a balanced binary sub-tree of replicas for the group; this sub-tree is called the Object Tree. An Object Tree logically represents the group membership of a single fault tolerant group.

In the event of a Registry failure, a new Registry is launched and retrieves fault tolerant group information from the object database. To further protect the system, each member of a fault tolerant group caches a local copy of information regarding the current status of the group memberships. This decouples the fault tolerant groups from the Registry, therefore, the system can continue to operate even in the absence of the Registry.

A set of IGOR objects harboring methods for fault tolerance are integrated into server applications, this alleviates the burden which would otherwise be placed on the server application to implement all fault tolerance mechanisms. These IGOR objects handle fault tolerant group membership related functionality, object monitoring and perform intragroup communication transparently from within the server application. IGOR objects have their own CORBA interfaces and converse amongst each other to perform maintenance tasks, such as reconfiguration and message propagation. Intragroup propagation uses the branches of the Object Tree (binary tree) as communication paths to ensure a single and complete group propagation of all messages. The Object Tree evenly distributes the burden of messaging to all group members. The responsibilities of transaction processing is shared between the IGOR objects and the server application. The Two-Phase Commit [3] protocol was used for the transaction processing associated with object state transfer among replicas.

A system should not render itself inoperable because of a partial network failures or a few downed computers. Fortunately, IGOR's redundant component design protects against such problems by automatically reconfiguring itself when failures occur. This is possible since the IGOR system is self monitoring and can quickly detect problematic objects and adjust accordingly.

4. Implementation Experience

Originally, the intention was to have the server application inherit fault tolerance mechanisms through a standard IGOR Object (C++ base class) with a CORBA interface. The inheritance approach proved to be infeasible due to some limiting constraints of the CORBA standard. Inheriting the IGOR Object class would entail inheriting both its IDL interface and the code associated with its fault tolerant mechanisms into a server application (which has its own IDL interface and associated code). In its current state, CORBA does not support multiple implementation interface inheritance within a single object. CORBA does support multiple interface inheritance within IDL, however, applications cannot inherit from multiple implementations of interfaces.

Consequently, we decided to incorporate the IGOR Object within the server application as a class member. Although not the original intent, inclusion of the IGOR Object as a class member yields a tightly coupled link between the server application and the IGOR Object. As a result, management of fault tolerant operations are largely performed by the IGOR Object on behalf of the server application, despite the inability to directly inherit such functionality. Coupling of the IGOR and server objects fuse the two objects together with interaction between them mediated by standard C++ method invocations. On the other hand, each object uses its own CORBA interface for remote communications.

As much fault tolerant code as possible has been offloaded to the IGOR Object. Unfortunately all the fault tolerant code cannot be handled by the IGOR Object alone. Certain aspects of server specific information (server's mutable state) must be handled by the server application in conjunction with the IGOR Object's involvement. The server's object state type cannot be known to the IGOR Object. Although the IGOR Object plays an important role in maintaining the data synchrony of the fault tolerant group, it does it somewhat blindly with respect to the actual data that is being transmitted. The IGOR Object knows nothing of the server's data, but it takes control of moving the data by instructing the server with respect to where to send or get state information. Again, intragroup state exchanges follow the branches of the Object Tree for communication.

Installing IGOR fault tolerance into a server application requires inheritance of a transaction processing class and creation of additional proxy methods to aid the IGOR Object in intragroup data transfer and transaction processing. Such proxy methods would not be required if direct object implementation inheritance was possible. Also, an object state class called StateKeeper is inherited, which contains methods for mutex locking and unlocking to protect the object's state from multiple thread access.

5. Conclusion

Conception of IGOR was a result of the need for fault tolerance in a standard operating environment. CORBA provides the means to realize fault tolerant distributed systems; IGOR capitalizes on the power and flexibility of CORBA to provide tools to aid in the creation of a fault tolerant system.

We were successful in completely isolating the client applications from all fault tolerance mechanisms by embedding fault tolerance functionality into the server applications, using pre-defined IGOR objects to perform a majority of the work.

Hopefully, CORBA's evolution will encompass the capabilities that make the design of fault tolerant systems less complex and less performance costly.

References

[1] Robert Orfali, Dan Harkey and Jeri Edwards *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc. 1996.

[2] Object Management Group *The Common Object Request Broker Architecture and Specification*, Revision 2.0, July 1995.

[3] Jean Bacon *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems*, Addison-Wesley Publishing Company, 1992.

[4] George Coulouris, Jean Dollimore and Tim Kindberg *Distributed Systems, Concepts and Designs*, Second Edition. Addison-Wesley Publishing Company, 1994.

[5] University of Tennessee *MPI: A Message-Passing Interface Standard*, May 5, 1994.

[6] Visigenic Software, Inc. *VisiBroker for C++ Programmer's Guide*, October 1996.

The Interception Approach to Reliable Distributed CORBA Objects *

P. Narasimhan, L. E. Moser, P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
priya@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

The Eternal system is a CORBA 2.0-compliant system that enhances the CORBA standard with replication and thus fault tolerance. The novel interception approach implemented in the Eternal system involves capturing IIOP-specific system calls made by the ORB, and subsequently mapping these calls onto a reliable multicast group communication system. The motivation for the use of this approach is that fault tolerance is transparent to the application objects, as well as to the ORB, and that any commercial ORB can be used with no internal modification. The interception approach exploits the performance of the underlying multicast group communication system to provide good performance.

1 Introduction

The incorporation of the object-oriented paradigm into the distributed computing model has resulted in the development of distributed object applications. Such applications must be portable, and the objects of the application must be able to interoperate when distributed across heterogeneous platforms with diverse hardware and software. The need for a standard that provides these features has led to the development of the Common Object Request Broker Architecture (CORBA).

While the CORBA standard provides for interoperability, language transparency, location transparency and portability, it does not address the issue of fault tolerance. Since there is an increasing need for reliable distributed object applications, current research is focusing on adding fault tolerance to CORBA.

*Research supported in part by DARPA grant N00174-95-K-0083 and by Sun Microsystems and Rockwell International Science Center through the State of California MICRO Program grants 96-051 and 96-052.

2 CORBA and IIOP

CORBA is a standard for communications middleware that defines interfaces to distributed objects and that provides mechanisms for communicating operations to objects by means of messages. The central idea of CORBA is the Object Request Broker (ORB), which mediates communication between client and server objects. All of the requests to, and responses from, the distributed objects are passed through the ORB.

To facilitate the interworking of commercial ORBs developed by different vendors, the CORBA 2.0 standard defines the Internet Inter-ORB Protocol (IIOP). IIOP allows objects operating over heterogeneous IIOP-compliant ORBs to interact with each other, irrespective of the internal structure of the ORBs or of any vendor-specific mechanisms. IIOP has a simple and generic interface that is designed to facilitate communication between hererogeneous ORBs. The IIOP-specific system calls invoked by the ORB are intended for the underlying TCP/IP layer.

3 The Eternal System

The Eternal system is a CORBA 2.0-compliant system that enhances the CORBA standard with fault-tolerance capabilities. Eternal exploits the facilities of an underlying multicast group communication system, in our case Totem, to provide CORBA-based applications with fault tolerance. In addition to providing reliable totally ordered multicasting of messages of the ORB, Totem provides mechanisms to deal with membership changes that occur when processors or processes fail, or the network partitions.

The Eternal system interfaces with the process group layer of Totem. The process group layer provides a simple set of group communication primitives and hides the implementation details of the underlying Totem protocols. Any multicast group communication system with an interface, membership services and guarantees similar to Totem, can alternatively be used.

4 Approaches to Fault Tolerance

Initial efforts to enhance CORBA with fault tolerance have taken an integration approach, with the reliability mechanisms incorporated into the ORB itself. With the advent of Object Services in the CORBA standard, other research efforts have taken a service approach, with the provision of a reliable object group service as part of the Object Services. To achieve the best of both of these previous approaches, we have adopted a novel "interception" approach.

These three different approaches are discussed briefly below and are illustrated in Figure 1. In all of these approaches, replication is employed to provide fault tolerance. The replicas of an object are considered to be members of an object group, where all of the replicas in the group have the same state. Requests can be conveyed to all of the replicas of an object by addressing the object group as a whole.

4.1 The Integration Approach

The integration approach [4], as implemented in the Electra ORB, as well as in Orbix+Isis, involves layering the ORB over a reliable ordered multicast group communication system. To enable the ORB to communicate its messages over the underlying system, adaptor objects are interpositioned between the reliable multicast system and the ORB. The mechanisms for the replication of objects and for the consistency of the replicas are embedded within the ORB, thus requiring internal modification of the ORB. The advantage of this approach is that it ensures transparency of the fault tolerance to the application objects since all of the necessary mechanisms are incorporated into the ORB itself. The application objects simply use the ORB as a communication path for their requests and responses.

4.2 The Service Approach

The service approach, as implemented in the OpenDREAMS project [2], involves providing an Object Group Service as part of the suite of Object Services that are defined by CORBA. The application objects convey their invocations and responses, via the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI), to their associated OGS objects, which then coordinate with each other to perform the operation on the replicas of the object and to return the results appropriately. The advantage of this approach is that it is wholly compliant with the CORBA standard and requires no proprietary mechanisms. However, the fault tolerance is now visible to the application objects since the application objects must be aware of the existence of the OGS objects in order to utilize their services.

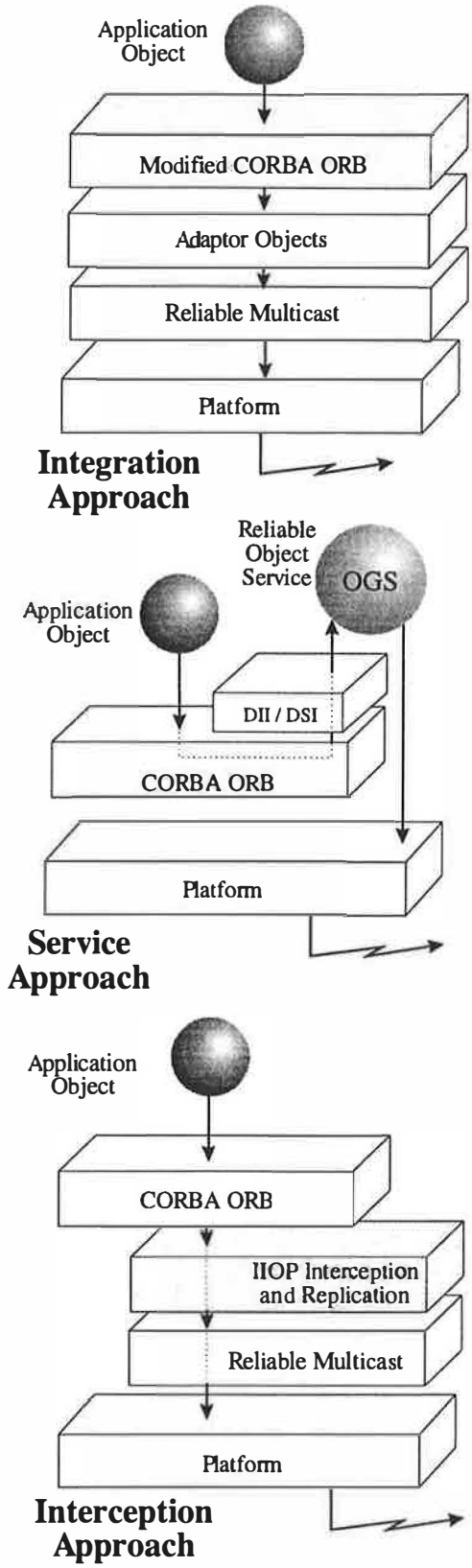


Figure 1: Different approaches to reliable CORBA.

4.3 The Interception Approach

The interception approach, as implemented in the Eternal system [7], involves capturing the system calls of the objects hosted by the ORB. The intercepted calls, which were originally directed by the ORB to TCP/IP, are now mapped onto a reliable ordered multicast group communication system. The advantages of this approach are that neither the ORB nor the objects need ever be aware of being "intercepted" and, thus, the fault tolerance is not visible to the application objects. Furthermore, the internal structure of the ORB requires no modification since the mechanisms that provide reliability are external to the ORB.

5 The Interception Approach in Eternal

5.1 "Catching" IIOP System Calls

Every CORBA object, on its creation, is associated with a unique Unix process identifier *pid*. Using user-level extensions [1] to the operating system, the system calls of the object can be traced using the file */proc/pid*, which is a part of the */proc* interface in Unix. The system calls of these objects can be monitored and captured. In addition, the arguments of these intercepted system calls can be modified before the calls are allowed to proceed to the operating system.

The Eternal Interceptor "catches" the calls made by the ORB, via IIOP, to TCP/IP. These system calls are then mapped onto the routines of the process group interface of the Totem system, which assumes the responsibility for multicasting messages. Of interest to us are only those system calls that are invoked by the ORB for establishing connections between objects and for maintaining the interaction of objects on these connections. Thus, the Eternal Interceptor catches system calls such as *open()*, *close()*, *read()*, *write()* and *poll()*, which involve TCP/IP connections and file descriptors. This specified set of system calls finds its correspondence in the set of routines of the process group interface of the underlying Totem system.

5.2 Replication of Objects

In the Eternal system, both client and server objects can be replicated. The object group abstraction of a replicated object enables any client object in the system to address the replicas of a server object as a whole, using a unique object group identifier. The translation of the object group identifier into the individual object references of the object group members is done transparently by Eternal.

The objects of Eternal in the CORBA space are in one-to-one correspondence with processes in the Totem framework. Eternal maintains the mapping between object groups and process groups, and extends the process group membership services of Totem to object groups.

Most importantly, Eternal ensures that the states of the replicas of an object remain consistent. The reliable totally ordered multicasts of Totem guarantee that the replicas of an object "see" the same operations in the same order. However, in a system where replication is employed, it is possible for duplicate invocations and duplicate responses of objects to occur. These can potentially corrupt the state of an object. Eternal provides mechanisms to detect and suppress such duplicate operations.

Eternal also manages the creation of new replicas and the removal of existing ones. It also undertakes the placement and distribution of replicas and handles the degree of replication of objects.

6 Benefits of the Interception Approach

6.1 Replication Transparency

Using the interception approach, Eternal captures the calls of an object and transparently maps these calls onto an object group. Thus, a client object is only ever aware of addressing a single server object while, in fact, the request is communicated to each of the server replicas. Similarly, a server replica is only ever aware of returning its results to a single object while, in fact, the results are returned to all of the client replicas.

Replication transparency allows the application developer to write an object-oriented program for the application as if it were to run on a single machine, rather than across a distributed system. Eternal assumes the responsibility of replicating and locating the application objects, and maintaining the consistency of the replicas of the objects across the distributed system.

6.2 Use with Commercial Off-the-Shelf ORBs

The interception approach allows Eternal to "attach" itself transparently to any commercial off-the-shelf implementation of the CORBA 2.0 standard. Thus, the ORB itself is never aware of its calls being traced, or of the interpositioning of Eternal between the ORB and the operating system.

This implies that any application operating on any commercial ORB could take advantage of the replication and fault tolerance capabilities of Eternal without any modification to the application code or to the internal structure of the ORB.

6.3 Use of the IIOP Interface

The Internet Inter-ORB Protocol (IIOP) is supported by complete implementations of the CORBA 2.0 standard. It has a simple and generic interface, which is designed to facilitate communication between heterogeneous ORBs.

Eternal captures the calls of the IIOP interface and maps them to Totem at the client, and receives the Totem multicast messages and maps them to IIOP at the server. Thus, it is possible for the client and the server objects to be hosted on entirely different ORBs, provided that these ORBs are equipped with IIOP. Thus, the replicas that constitute an object group could, in fact, be objects implemented in different languages and running over different ORBs. The only stipulation is that these objects be able to communicate over IIOP. Fortunately, an increasing number of vendors now supply IIOP as their native protocol.

6.4 Performance

The Eternal system is currently under development, using various implementations of the CORBA 2.0 standard that are commercially available, including the CORBA-compliant Inter-Language Unification (ILU) [3] from the Xerox Palo Alto Research Center.

A typical application using a single server object and a single client object over ILU without Eternal involves 910 object invocations per second. With three-way replication of the same client and server objects over ILU with Eternal, preliminary measurements yield results of 670 object invocations per second. These measurements indicate that the overhead associated with interception and multicasting is not unreasonable for replicating objects, particularly since the code of the ORB and the operating system are unmodified. With further optimization of the code of Eternal, we anticipate even better performance.

Since replication of objects requires interaction between groups of replicas, some sort of underlying multicast group communication is required. The use of a reliable totally ordered multicast group communication system simplifies the ordering of operations at the replicas, and yields better performance than multiple point-to-point TCP/IP connections between each pair of interacting replicas. The high performance of the underlying Totem multicast group communication system is exploited by Eternal to obtain good performance.

7 Conclusion

Eternal enhances the CORBA standard by allowing application objects to be replicated and distributed on different machines across the system, while maintaining consistency of the replicas of the objects. Replication of objects across the distributed system provides tolerance to a variety of hardware and software faults. It also allows hardware and software components to be replaced while the system is live so that the application can continue to operate without interruption of service.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser and C. J. Scheiman, "Extending the operating system at the user level: The Ufo global file system," *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA (January 1997), pp. 77-90.

[2] P. Felber, B. Garbinato and R. Guerraoui "Designing a CORBA group communication service," *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara on the Lake, Canada (October 1996), pp. 150-159.

[3] B. Janssen, D. Severson and M. Spreitzer, ILU 1.8 Reference Manual, Xerox Corporation (May 1995), ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[4] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, John Wiley & Sons Publishers, New York (1997).

[5] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4 (April 1996), pp. 54-63.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification* (1995), Revision 2.0.

[7] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Exploiting the Internet Inter-ORB protocol interface to provide CORBA with fault tolerance," *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, Portland, OR (June 1997) (this volume).

NOTES

NOTES

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of engineers, scientists, and technicians working on the cutting edge of the computing world. The USENIX technical symposia and system administrator conferences are the essential meeting grounds for the presentation and discussion of the most advanced information on the developments of all aspects of computing systems.

USENIX and its members are dedicated to:
- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's bi-monthly newsletter featuring technical articles, system administration tips and techniques, SAGE News, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as nine technical meetings every year.
- Discounts on the purchase of proceedings from USENIX conferences and symposia and other technical publications.
- Discount on purchases of USENIX CD-ROMs.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Nolo Press, O'Reilly & Associates, Prentice Hall, Sage Science Press, and John Wiley & Sons.
- Special subscription rates on periodicals: *The Linux Journal,* UniForum's *IT Solutions*, and the annual *UniForum Open Systems Products Directory.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Adobe Systems Inc.
Advanced Resources
ANDATACO
Andrew Consortium
Apunix Computer Services
Boeing Commercial
Crosswind Technologies, Inc.
Earthlink Network, Inc.
ISG Technologies, Inc.
Matsushita Electric Industrial Co., Ltd.
Motorola Research & Development
MTI Technology Corporation
Sybase, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

Sage Supporting Members:

Atlantic Systems Group
Bluestone, Inc.
Enterprise Systems Management Corp.
Great Circle Associates
OnLine Staffing
Paranet, Inc.
Pencom Systems Administration/PSA
Southwestern Bell
Taos Mountain
Texas Instruments, Inc.
TransQuest Technologies, Inc.

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*
URL: *http://www.usenix.org*

ISBN 1-880446-86-3